DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/88S-1

DESIGN OF A SYNTAX VALIDATION TOOL FOR REQUIREMENTS ANALYSIS USING STRUCTURED ANALYSIS AND DESIGN TECHNIQUE (SADT)

THESIS

Dong Hak Jung
Major, ROKAF

AFIT/GCS/ENG/88S-1

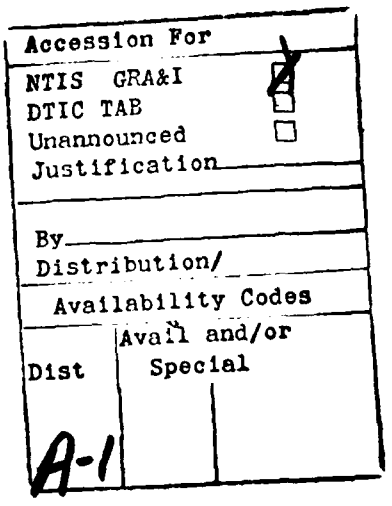
Approved for public released; distribution unlimited

AFIT/GCS/ENG/88S-1

DESIGN OF A SYNTAX VALIDATION TOOL FOR REQUIREMENTS ANALYSIS USING STRUCTURED ANALYSIS AND DESIGN TECHNIQUE (SADT)

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer System



Dong Hak Jung, B.S.
Major, ROKAF

September 1988

Approved for public released; distribution unlimited

Preface

This report documents my thesis effort to design and implement a validation tool which checks the syntax of the Structured Analysis and Design Technique (SADT) method from a structured analysis diagram.

I would like to express my sincere appreciation to Dr. Gary B. Lamont, my thesis advisor, for all his guidance and inspiration throughout this effort. I would also like to thank my committee members, Dr. Thomas C. Hartrum and Capt. David W. Fautheree, for their contribution to this thesis. I would also like to thank Maj. Bill E. Oswald and his family for encouraging me and for English reviews of this thesis. I would also like to thank the U.S. and R.O.K. Governments for allowing me to have this opportunity.

Finally, I would like to thank my wife, [redacted] and our [redacted] children, for their understanding and moral support during our AFIT assignment.

Dong Hak Jung

Table of Contents

List of Figures

List of Tables

Abstract

This thesis investigation presents the prototype development of a validation tool for checking the syntax of Structured Analysis and Design Technique (SADT) method from a structured analysis diagram. The tool provides the requirements analyst and the designer with an environment for checking the SADT syntax of an SADT diagram.

The tool is operated through the use of an SADT Editor which was developed by Steven E. Johnson at the Air Force Institute of Technology (AFIT).

The validation tool was developed in three phases. During the first phase, the formal definition of the SADT graphical language was derived using Predicate Logic representation. During the second phase, the SADT Editor was analyzed and the interface issues with the software were identified. Thus, the graphical features were translated. During the third phase, the syntax rules were identified according to the formal definition of the SADT methodology using Predicate Logic representation.

The new tool was implemented into a knowledge-based system to ease the extension of the syntax rules, to add knowledge of the SADT graphical structure and to add domain knowledge of an application system developed by the SADT methodology.

DESIGN OF A SYNTAX VALIDATION TOOL FOR REQUIREMENTS ANALYSIS USING STRUCTURED ANALYSIS AND DESIGN TECHNIQUE (SADT)

I. Introduction

Objective

The objective of this thesis effort is to perform the prototype development of a syntax validation tool for graphical diagrams using Structured Analysis and Design Technique (SADT) (SADT is a trademark of SofTech, Inc). These diagrams could be used in the requirement and design analysis phase of the software life cycle (16:1-1). While editing a SADT diagram, the tool should be able to check whether or not structured analysis diagrams are valid for the SADT's syntax, produce error messages, do error recovery, and perform editing suggestions. Thus, this tool must have the knowledge of the SADT's syntax and an associated formal process for transforming SADT's graphical representation.

Background

Software Life Cycle. To illustrate the software life cycle, the "waterfall model" or "conventional life cycle model" has been proven convenient (3). Figure 1.1 shows the conventional software life cycle. The life cycle is divided into five phases, which are the requirement analysis phase, the design phase, the implementation phase, the test phase, and the maintenance phase.

During the requirement analysis phase, analysts try to understand the user's requirements and define the specifications to meet those requirements. User's specifications contain why the system is to be designed, what the system should do,

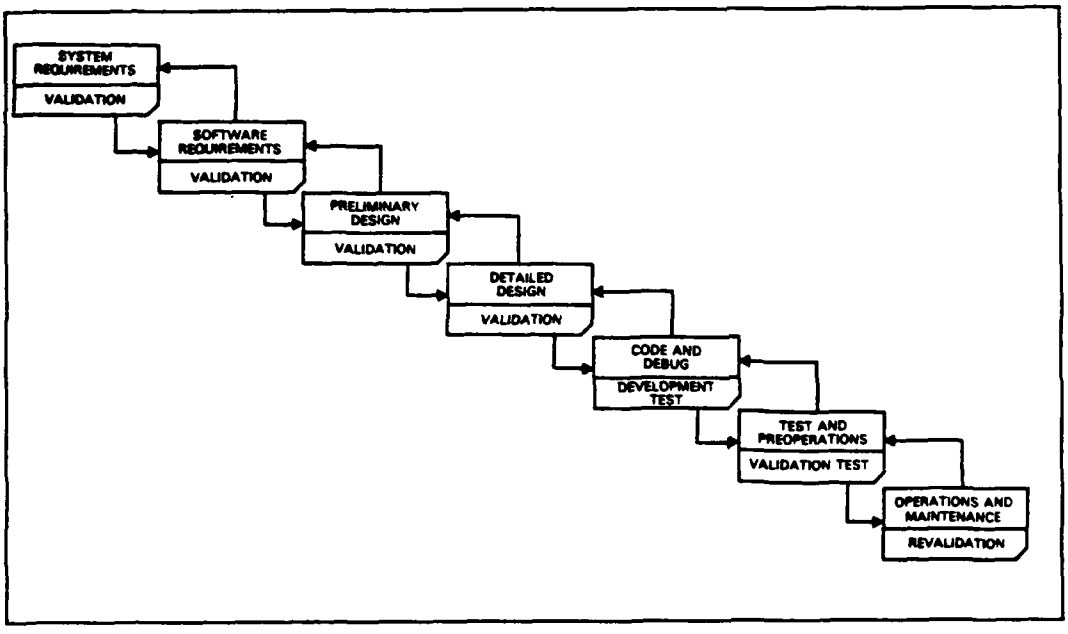


Figure 1.1. Conventional Software Life Cycle (1:3)

and what design constraints are to be considered. During this phase, the software specifications should be determined for satisfying these requirements.

The design phase specifies how the system is to be implemented so that it meets the software specifications derived from the requirement analysis. Preliminary design and detailed design are the two steps performed in the design phase. The preliminary design is concerned with the transformation of the software specifications of the previous phase into specific design components. The components may be further decomposed into sub-components as necessary. Thus, components and sub-components are realized in terms of functional modules within a hierachical description or framework.

The detail design focuses on refinements to the architectural representation that leads to detailed data structure and algorithmic representations for each component.

The implementation involves coding of each module using a formal computer programming language.

During the test phase, each program module is tested in order to find and correct the errors. Then, an integration test is performed to merge each program module into a whole hardware/software system.

Finally, during the maintenance phase, the system is operated and modified as necessary.

Some critics claim that the conventional software life- cycle model is not effective for the process of preparing detailed software specifications because it is difficult to separate the "what" of specification from the "how" of design (1).

Alternative software development methods (or paradigms) have been suggested for overcoming drawbacks of the conventional software life cycle model (2). The new paradigms are "transformation systems", "prototyping", and "operational specification".

The "transformation systems" paradigm is shown in Figure 1.2.

> The "transformation systems" paradigm uses automated support to apply a sequence of correctness preserving transformations to a formal specification. The transformations reduce the high level constructs of the formal specification into lower level constructs (such as data structures and algorithms) which form a software system. Additionally, the sequence of transformations is recorded. This allows maintenance to be performed by simply modifying the specification and repeating the transformation process guided by the previously recorded sequence of transformations [2].

The "prototyping" paradigm is shown in Figure 1.3.

> This paradigm uses the system requirements to construct a prototype of the desired software system. The objective of the prototyping effort is to clarify the characteristics and operation of the system by constructing a version that can be exercised. The prototype then provides a vehicle

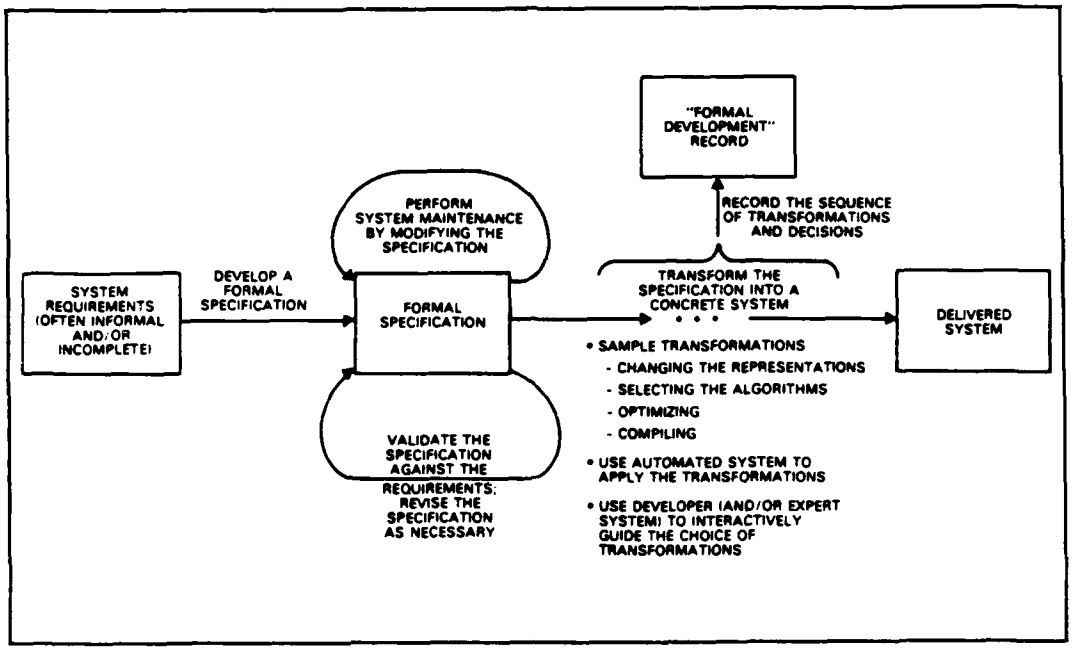


Figure 1.2. Transformation System Paradigm (2:9)

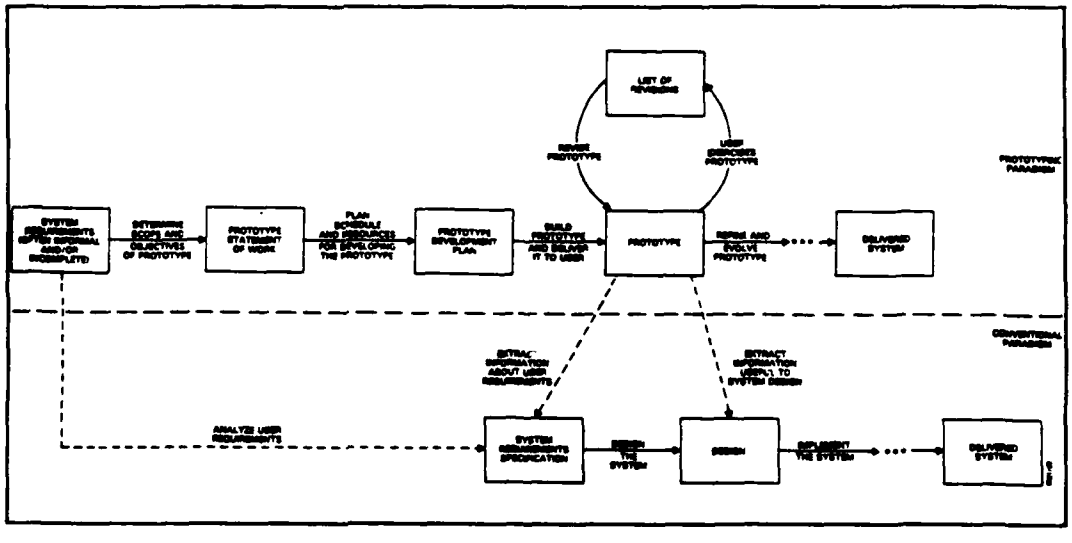


Figure 1.3. Prototyping Diagram (2:7)

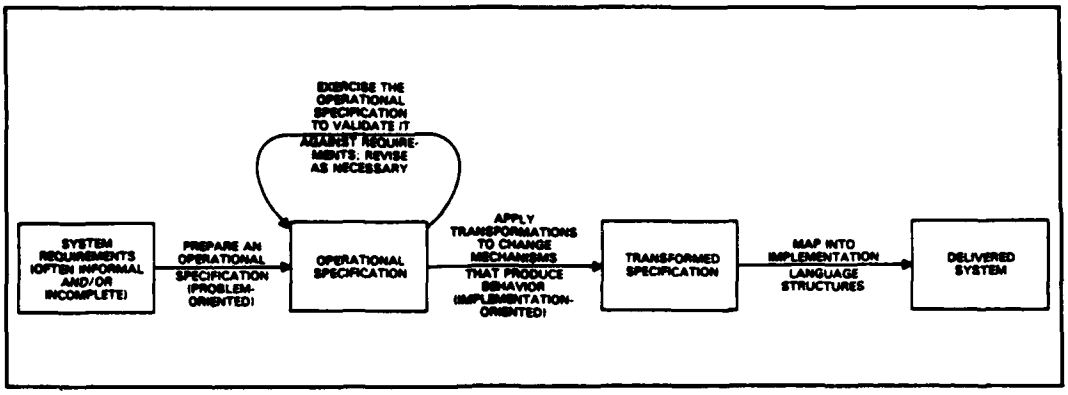


Figure 1.4. Operational Specification Paradigm (2:8)

by which both the user and system designer can evaluate the development of the system. Based on the evaluations, the prototype is refined until a complete system requirements specification has been defined. At this point, development of the software could continue by: developing the prototype into an operational system, going to the design stage of the conventional life-cycle model, or using the transformation system paradigm [2].

The "operational specification" paradigm is shown in Figure 1.4.

In this paradigm, an operational specification is used to generate a system model that can be executed to examine the behavior of the system (much like the "prototyping" paradigm). This approach acknowledges the interweaving of the "what" and "how" considerations with the goal of producing an operational specification that deals only with problem oriented issues, and the operational specification is expressed in some language that allows it to be executed to examine system behavior. Once the operational specification has been completed, a transformation system is used to obtain an actual software system [2].

SADT. Structured Analysis Design Technique (SADT) is SofTech's methodology for guiding requirement analysis and system design of the software life cycle (14).

> Originally introduced as a "system-blueprinting" method for documenting the architecture of large and complex systems, SADT had become a full-scale methodology for coping with complexity through a team-oriented, organized discipline of thought and action, accomplished by concise, complete, and readable word and picture documentation [13:25].

Specifically, SADT provides techniques and methods for:

1. Thinking in a structured way about large and complex problems;
2. Communicating analysis and design results in clear, precise notations;
3. Controlling accuracy, completeness, and quality by procedures for review and approval;
4. Documenting the system analysis and design history, decisions, and current results;
5. Working as a team with effective division and coordination of effort; and
6. Managing development projects and assessing progress [16:1-1].

SADT provides "both techniques for performing systems analysis and design and a process for applying these techniques which significantly increases the productivity of a team of analysts or designers" (16:1-1).

SADT provides a graphical technique which models a problem to be solved into the box-and-arrow diagrams. In addition to boxes and arrows, each diagram is described with some text for understanding. Therefore, an SADT model consists of diagrams and text derived from a graphical language.

Since the graphic language consists of the notations for structured analysis, the resulting diagrams are well-organized structurally and hierachically. SADT is based upon a maxim that "everything worth saying about anything worth saying something about must be expressed in six or fewer pieces" (14:26).

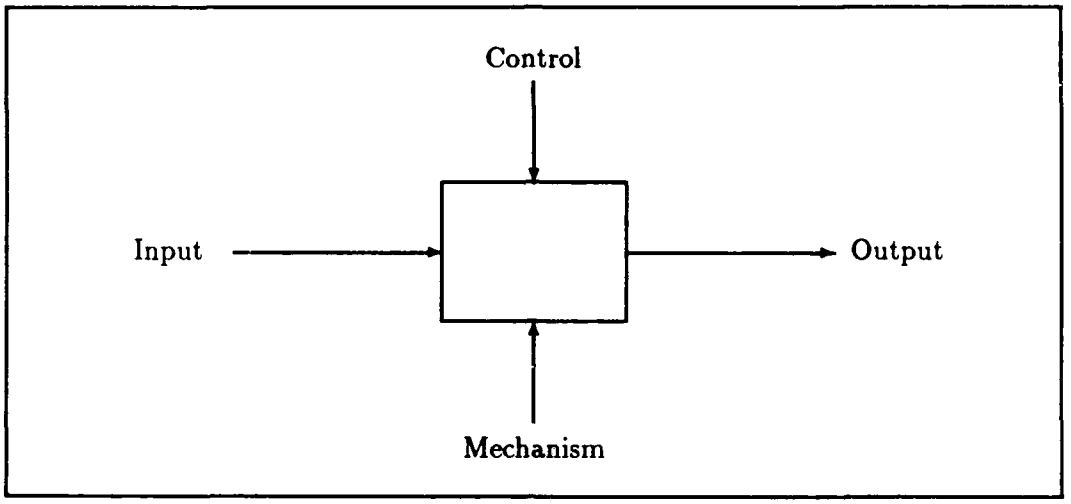


Figure 1.5. SADT Digram

> The maxim implies a hierachical, top-down decomposition of the whole into easy-to-grasp chunks, and in the process, the whole and all of its subwholes and parts become more understandable because each whole bounds the context within which its parts are to be understood [14:26].

The maxim being applied to the graphical notations of structured analysis, each one of the six or fewer pieces is uniformly expressed in the form of the box, whose four sides always mean input, control, output, and mechanism, as shown in Figure 1.5.

Figure 1.5 indicates that the input is transformed into the output. The control defines under what conditions the transformation occurs, and the mechanism defines how the function is physically accomplished.

An SADT diagram consists of boxes and arrows with some text. Arrows, which are input, control, and output, connect boxes together and represent interfaces or interconnections between the boxes. The interfaces are indicated by branching arrows that connect outputs to inputs or controls (and sometimes mechanisms). The

result of one transformation can control the transformation of some other input by another box or can be further transformed by another box.

The SADT diagrams of a problem are the decomposition of a bounded subject. Subjects are a box and the arrows that touch it. The diagram that contains the boundary is called the "parent" diagram. The diagram that decomposes one box on the parent diagram is called the "child" diagram.

Boxes are named and arrows are labeled. Boxes are numbered and arrow ends may be tagged with ICOM (standing for INPUT, CONTROL, OUTPUT, MECHANISM) codes. A number follows the letter I, C, O, or M sequentially top to bottom or left to right. ICOM codes provide the way to quickly verify whether or not the external arrows of a diagram match the boundary arrows of the corresponding box on the parent diagram. They also ensure consistent decomposition, since one must account for all arrows entering and leaving a diagram in a low level diagram.

A collection of diagrams for a problem is called a "model". SADT provides "the same graphic notation for both the things and the happenings aspects of any subject" (16:19). Every model has two dual aspects- a thing aspect, called data model and a happening aspect, called activity model.

In activity models, box names are verb phrases describing the activities, and arrow labels are noun phrases describing the data involved in the activity. Thus, the things are transformed by the happenings. Data models result from an opposite approach: box names are noun phrases describing the data, and arrows labels are verb phrases describing the activities involving the data.

Syntax-Directed Editor. Syntax-directed editors are editors which use the syntax of the programming language while editing a program. While text editors treat programs as text, syntax-directed editors use the underlying syntax of the programming language. When syntax-directed editors are used for editing a program, the program is built from the syntactic elements of the language. Most syntax-directed

editors use templates to build programs. Templates are invoked by the user. Templates are predefined patterns of code which consist of keywords, punctuation, and nonterminals. The nonterminals are placeholders which the user fills in with templates or typed code.

Syntax-directed editors provide an environment which enhances the productivity of both beginning and experienced programmers (18). Programmers do not need to remember the entire syntax of a programming language when using a syntax-directed editor. Thus, programmers benefit by the typing time saved and the immediate detection of syntax errors. Programs written using syntax-directed editors are well-formatted, readable, and syntactically correct.

There are many syntax-directed editors and programming language environments. A well-known syntax-directed editor is the Cornell Program Synthesizer (19). It is an interactive programming environment, designed primarily as a teaching tool. It includes a syntax-directed editor, a compiler, and a debugger. The first language for the Cornell Program Synthesizer was PL/CS, which is a subset of PL/1. Currently, it employs PASCAL. Comprehensive descriptions and a bibliography on the syntax-directed editors and programming environments is provided in (9).

Graphics Language Translation. Since SADT is a graphical language, all information on the SADT diagrams should be translated into well-defined descriptions.

Douglas T. Ross, who was the developer of the SADT, said,

> Although it has yet to be formalized, SA as a modeling language is both rigorous and complete. Even the combination of SADT with the RML language of requirements modeling, described by Greenspan (5), is the association of a particular formal semantics for an interpretation of an SADT model's syntax rather than formalization of SA semantics itself, as Greenspan acknowledges. Formalization of SA itself is very difficult [13:28].

Requirement Modeling Language (RML) is a language which provides a way to model "real-world" problems during the requirement analysis phase of the software

development (5). Greenspan proposed that requirement modeling be divided into two steps:

1. Structured modeling (using boxes and arrows): Decide what are the relevant concepts, decide on what to call them, organize into a "model" by concerning graphically.
2. Semantic modeling: Create a generate object in RML for each concept named in the SADT model; give object definitions in RML [5:77].

He used the SADT method as an intermediate step for modeling software requirements into RML language. He has also shown that a derivation of an RML model from an SADT description can be relatively straightforward. In addition, RML is supported with a formal definition using a First-Order Logic (FOL) with time (5:27-41).

PSL/PSA (Problem Statement Language / Analyzer) is another requirement specification language (17). The language, PSL, provides object types, such as INPUT, PROCESS, OUTPUT, SET, ENTITY, INTERFACE, etc., which together with several relationship types allow statements. The analyzer, PSA, checks for certain kinds of inconsistencies, such as invalid combination of object/relationship types or the omission of mandatory relationships, and allows various reports to be extracted (17).

Another effort to translate a graphics (dataflow) diagram into a function has been attempted (12). This effort resulted in a process for translating dataflow components into design schemas. For example, suppose a desigr schema has three inputs from the domain A, B, and C and generates two outputs from the domain H and I. Thus, the design schema can be represented as a function, f: (A*B*C) $\rightarrow$ (H*I). This function can also be represented as a dataflow diagram with three inputs and two outputs.

Problem and Scope

Early uses of SADT were performed with pen and paper, resulting in a lack of standards and the possibility of inaccurate data. Although several graphic computer support tools for SADT have been developed as AFIT thesis investigations (21) (8), there are still the possibilities of diagram errors. Also, an automated interactive system, AUTOIDEF (Automated ICAM (Integrated Computer-Aided Manufacturing) Definition), supports a graphic tool for SADT (17). However, these tools do not have the procedures for checking the syntax of SADT.

This thesis effort will attempt to solve these problems through the the development of a validation tool for checking the consistency of SADT. Also, it will provide error messages, error recovery, and editing suggestions.

Assumptions

Since a graphic tool for the SADT method is available at AFIT, it is reasonable to use this tool for this thesis effort. The specific tool selected was developed by Steven E. Johnson (8). Since he used the Sun 3 (Sun is a trademark of Sun Microsystems Inc.) workstation for his SADT tool, the implementation of this tool will be performed on the Sun 3 workstation using Berkely UNIX (UNIX is a trademark of AT and T) Version 4.2, because there are several Sun 3 workstations in the Information System Laboratory of AFIT. These workstations also provide a graphics software SunView and the Sunwindow environment.

The user of the tool is assumed to be familiar with SADT.

Approach

First, the SADT language is analyzed with emphasis upon its syntax and graphic features. Thus, the syntax and the graphic features are formalized in order to have a precise meaning of "consistent" and "well-understood". The formalization for SADT could be made using logic, algebraic, function, and other formal approaches.

Secondly, the formal definitions of SADT's syntax and graphic features are transformed into formal computer language forms. Several computer languages such as Prolog, Lisp, Ada, etc. could be useful for this purpose. The resulting forms will become a data base for validating SADT diagrams.

Thirdly, the control procedure should be developed for checking SADT diagrams for validity in relationship to a specified data base.

From the above discussion, an expert system could be used for the implementation of this thesis investigation. The data base derived from SADT's syntax and graphic features becomes the knowledge base for the expert system and the control procedure becomes the inference engine of the expert system. This expert system can be extended with SADT's semantic and domain knowledge of the application systems using SADT methodology.

Also, the whole system will be operated interactively for syntax checking, producing error messages, error recovery, and editing suggestions.

Sequence of Presentation

This thesis consists of five chapters. The requirements of the formal definition of SADT and the syntax validation tool are defined in Chapter II. Based upon the requirement analysis, the SADT graphical structures are formalized, and the tool is designed in Chapter III. Chapter IV presents the selection of a formal computer language depending upon the formalization of SADT for coding, and the tool is implemented and tested. In Chapter V, the conclusions and the recommendations are discussed for future investigations.

II. Requirement Analysis

Introduction

This chapter presents the requirements for the SADT validation tool. The issues to be considered are existing constraints related to this tool, hardware and software support, human interface, formalization criteria, the functional model of this tool, and evaluation criteria.

Existing Constraints

The current computer graphic tool for SADT available at AFIT was developed by Steven E. Johnson (8). This tool provides an interactive graphics editor for SADT diagrams. A summary of his tool is presented in Appendix A. The tool provides the means to generate data dictionary information (8:3-2). However, the tool does not provide the capability to check the SADT's syntax in any SADT diagrams. Thus, the new tool to be developed in this investigation should interface with Johnson's tool by providing the capability to check the SADT's syntax. Also, the new tool must be compatible and satisfy all requirements of the previous SADT tool (8).

Hardware Support

Since this tool must be integrated into AFIT computing environment, hardware restrictions imposed by that environment must be considered when implementing the new SADT tool. The AFIT's computing environment provides two central computers, which are the VAX 11/785 (VAX is a trademark of Digital Equipment Corporation Inc.), and several stand-alone workstations. The workstations, Z-100 and Z-248 Zenith microcomputers, access the central computers through the use of the AFITNET. An Ethernet communication package in the AFITNET also allows for remote login on the SUN workstations directly to the central computers.

Johnson's tool was developed using the SUN workstation. Thus, it is desirable to develop the new SADT tool on the SUN workstation because many portions of Johnson's tool can be reused without modifications.

Software Support

Software support needed in the development of this tool is not as simple. After formalization of the SADT's syntax, a decision must be made concerning which computer language can be used to write a formal definition of the SADT's syntax properly. The computer language selected should interface with Johnson's tool. His tool was written into C language, and uses graphics software package called SunView and the Sunwindow window environment (8). Thus, this computer language must be able to interface with the C language and the graphics software SunView. Minimal execution time is also a desired feature.

Human Interface Requirements

A computer system's effectiveness is directly related to how well the system was developed so that users can use it easily. James. W. Urscheler, in his master's thesis *Design of a Requirement Analysis Tool Integrated with a Data Dictionary in a Distributed Software Development* (21), presented five key psychological factors to be considered for design of an effective system. The five key factors are:

1. Keep the user motivated - do not frustrate or bore him.
2. Break the lengthy input process into parts to permit the user to achieve "psychological closure". This provides positive feed back to the user through a feeling of accomplishment and success.
3. Minimize the memorization required by the user.

4. Provide visually pleasing displays on the screen. This includes minimizing the scrolling and other distracting movements of text, the highlighting of instructions to the user, and making effective use of margins and white space.
5. Keep response time to a minimum. Display status messages to keep the user constantly informed of what is happening inside the machine [21:21].

In addition to these factors, error recovery guidelines and user prompts should be provided on screen.

Each of these human interface requirements should be addressed during the design and the implementation phase of the tool development.

Formalization Criteria of SADT

From the previous chapter, the formalization of SADT was necessary to check the errors in the SADT diagrams. Formalization of SADT should support the following requirements:

1. Formal definition must contain the syntax information in any SADT diagram and be described syntactically.
2. Formal definition must provide the means to determine syntax errors in any SADT diagram.
3. Formal definition should provide a domain where the definition of "consistency" can be given.
4. Formal definition should serve as the final arbiter in cases where there is disagreement concerning the exact meaning of the representation.
5. Formal definition should be able to be implemented in a computer system.

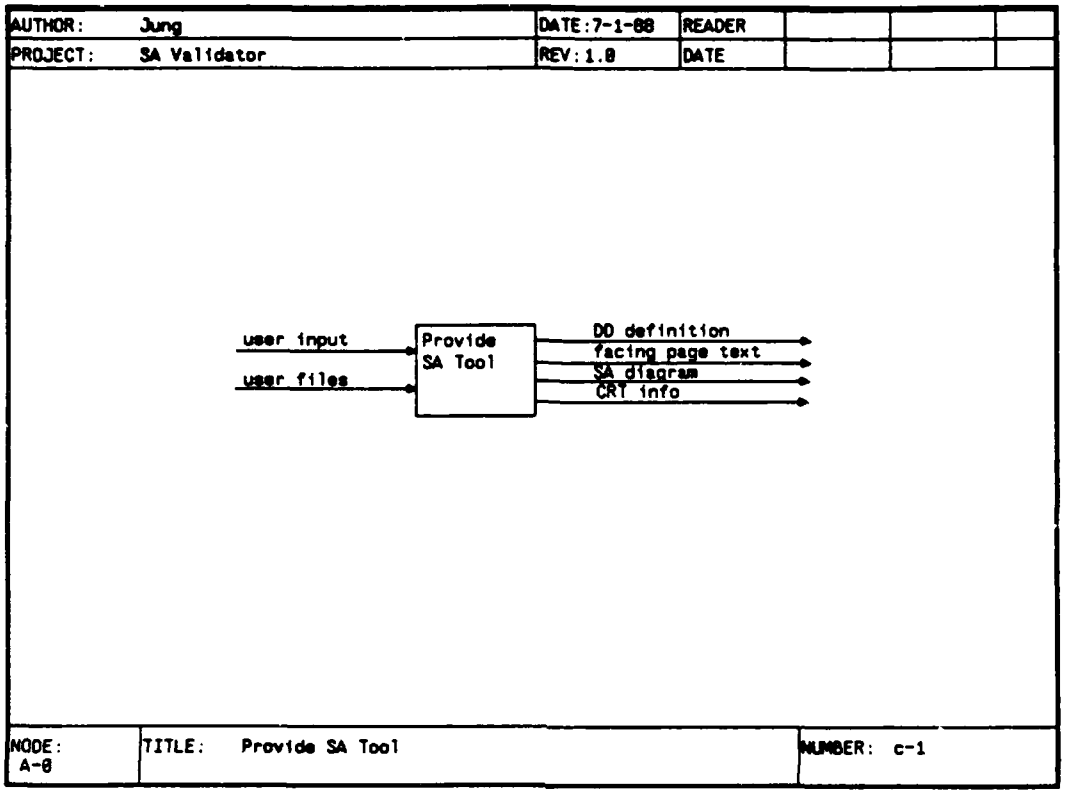


Figure 2.1. Top Level of SADT Tool

Functional Model for SADT Validation Tool

This section presents a functional model which defines and describes the tool's functional requirements discussed in the previous sections. The following figures and discussions display and explain the SADT diagrams associated with the higher levels of the functional model for the SADT validation tool.

Figure 2.1 displays the top level of the tool's functional model. This diagram indicates the overall requirements of the SADT tool. The "Provide SADT Tool" operation is the process which creates and edits an SADT diagram and its data dictionary information from tool user. Also, facing page text for the data dictionary is generated (8). In addition, the operation contains the process which checks the SADT's syntax of the SADT diagram and produces the syntax error-free SADT

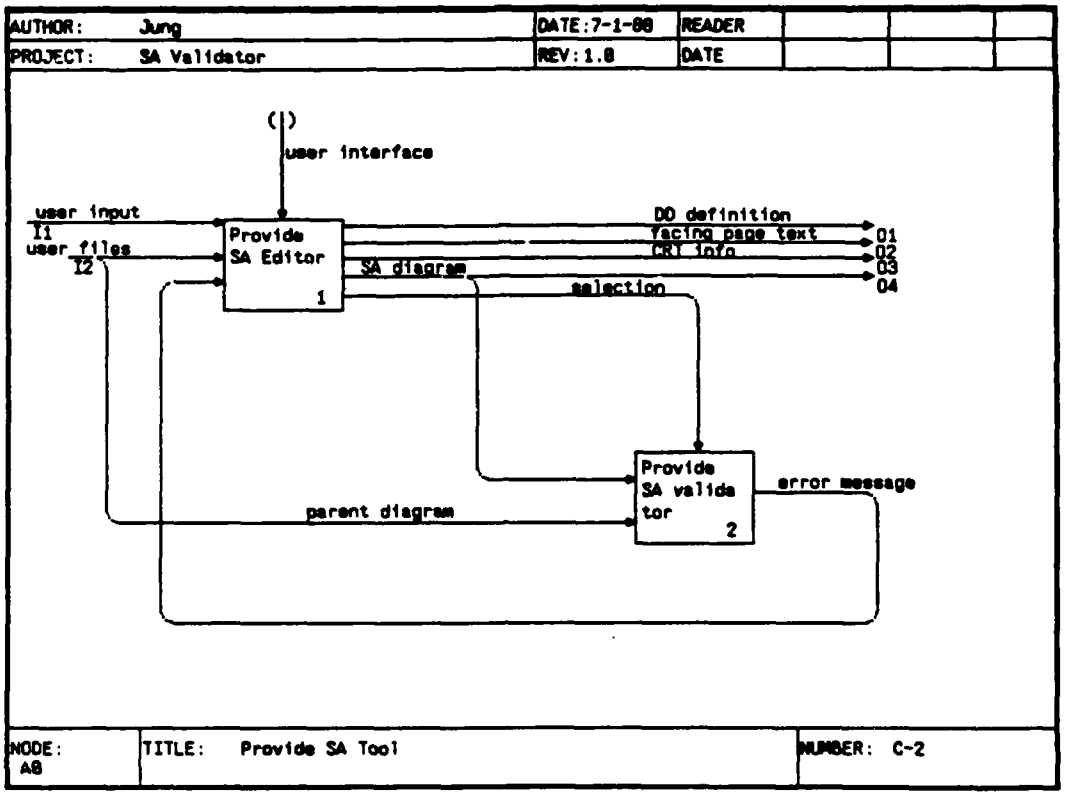


Figure 2.2. Provide SADT Tool

diagram.

Figure 2.2 displays the initial decomposition of the top levels of the functional model. This decomposition indicates the two primary functions or components of the SADT tool. The "Provide SADT Editor" operation is the process which edits and produces an SADT diagram and its data dictionary information, and facing paging text. This operation has been developed by Johnson (8). The "Provide SADT Validator" operation is the process which checks the syntax errors of the SADT diagram produced from the "Provide SADT Editor" operation and produces error messages as necessary.

Figure 2.3 shows the decomposition of the "Provide SADT Validator" operation into its component functions. The "Translate Diagram" operation is the process

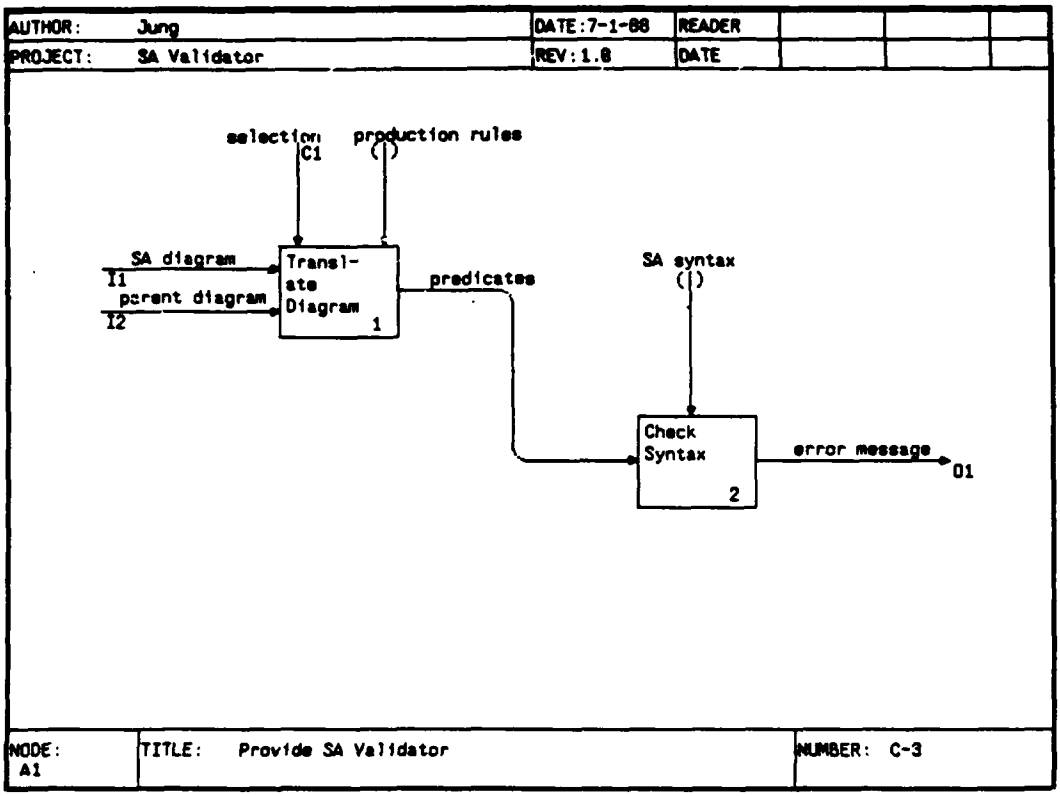


Figure 2.3. Provide SADT Validator

which translates an SADT diagram into formalized forms using the translation rules. The "Check Syntax" operation is the process which checks the SADT syntax using the syntax rules and produces error messages as necessary. Further decompositions of these two operations are presented in Appendix B.

Evaluation Criteria

In order to measure the success of the SADT validation tool in meeting its requirements, a set of evaluation criteria must be established. Several parameters can be used to measure to success of the tool.

The most important parameter measured is how accurately the tool checks the SADT's syntax error, and provides error messages and editing suggestions. The

other parameters to be considered are the average time spent to learn the tool, user friendliness, and the system responsiveness.

Summary

This syntax validation tool is conceived to support the requirement analysis phase of the software life cycle. Since this tool should extend Johnson's tool, this tool should satisfy all capabilities of his tool.

In order to check the SADT's syntax in an SADT diagram, all SADT's syntax and graphic features should be precisely defined during the formalization process. These requirements were presented in the formalization criteria section.

In addition, this tool should provide error messages, error recovery functions, and editing suggestions. These because this tool is to interface with Johnson's tool. Thus, the Sun workstation and the graphics software SunView are needed for the development effort.

III. Conceptual Design

Introduction

In this chapter, the design of the SADT validation tool is described and justified. The description begins with consideration of several constraints discussed in the previous chapter. Then, the overall functional system design is introduced. Also, the main functions of the SADT validation tool, translating and syntax checking process are presented. The SADT validation tool is called the SADT Validator and Johnson's SADT graphic editor is called the SADT Editor.

Constraints

Hardware and Software. Since the SADT Validator should interface with the SADT Editor, the hardware and the software to be used are already chosen. Thus, the Sun 3 workstation and the graphic software SunView are required for developing this SADT Validator. Also, since the SADT Editor was implemented using the C language, a decision was made to proceed using the C language for the translating process. This choice was very reasonable because many portions of the SADT Editor could be directly reused without modifications. Also, the software needed in the development of the syntax checking process could easily interface with the C language. A detailed discussion of this process is presented in Chapter IV.

Human/Computer Interface Constraints. As discussed in the previous chapter, an acceptable human/computer interface should be considered in the design phase. Especially, the design decision about the window manipulation should be addressed. Screen layout of the SADT Editor should be slightly modified adding a new menu item for the validating function of the SADT Validator. Figure 3.1 shows the modified screen layout for the SADT Validator.

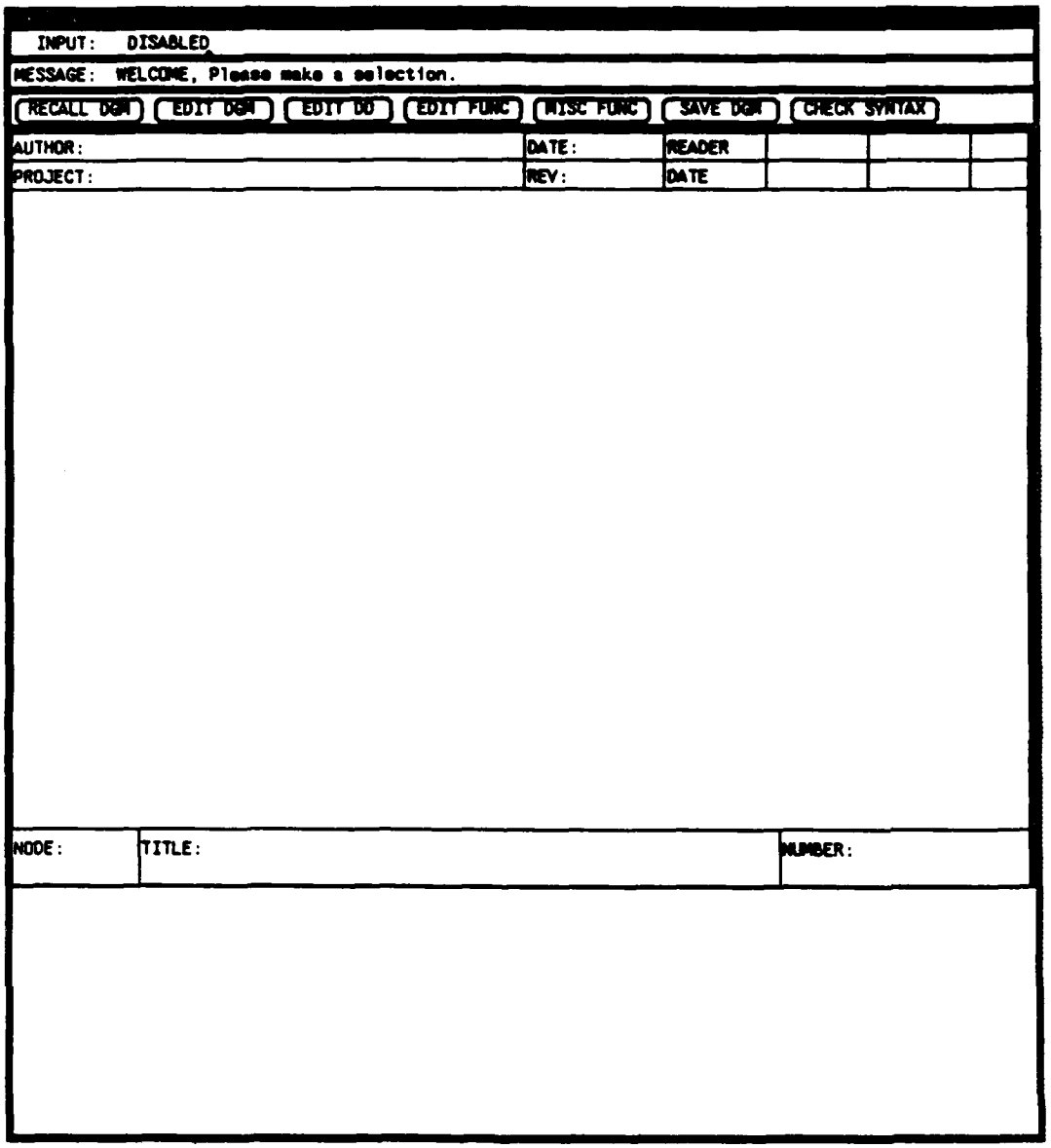


Figure 3.1. Screen Layout

There are five windows on the screen: the Input Window, the Message Window, the Selection Window, the Diagram Window, and the Data Dictionary/Syntax Error Message Window in a vertical order.

The functions of the Input Window, the Message Window, and the Diagram Window are the same as in the SADT Editor'. The third window, the Selection Window, is used for selecting the menu which users desire to operate. There are six ovals on the Selection Window: RECALL DGM, EDIT DD, EDIT FPT, EDIT FUNC, SAVE DGM, and CHECK SYNTAX. The RECALL DGM oval is used to read an existing diagram file. The EDIT DGM oval is used to create and edit a diagram. The EDIT DD oval is used to create and edit data dictionary information. The EDIT FPT oval is used to edit facing page text of a diagram. The EDIT MISC oval is used for miscellaneous functions such as, making a dump file for a diagram, exiting the SADT TOOL, etc.. The SAVE DGM is used to save the current diagram. Finally, the CHECK SYNTAX oval is used to check the SADT syntax.

The Data Dictionary/Syntax Error Message Window is used for two functions. One function is to enter the data dictionary information which cannot be accessed from the diagram. The other function is to display syntax errors of the diagram. The detailed description of the SADT Editor's screen layout is found in Appendix A.

System Structure

The overall system structure of the SADT Validator is based upon the components identified in the requirements discussed in the previous chapter. Figure 3.2 shows the overall system structure of the SADT Validator.

This system structure is directly produced from the SADT diagrams for the SADT Validator (see Figure 2.2 and Figure 2.3). There are six components in Figure 3.2: the SADT Editor, the Translator, the Syntax Checker, the Diagram files, the Translation Rules, and the Syntax Rules. This section examines each of these

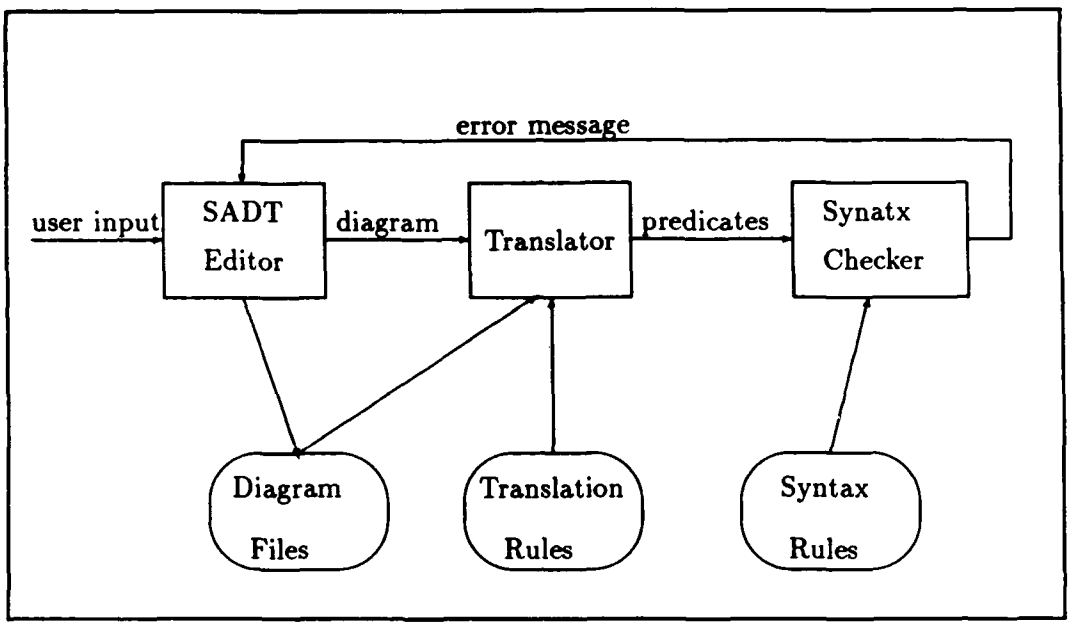


Figure 3.2. Overall Structure of SADT Validator

components based upon their functions in the system as well as their relationships to produce the overall system structure.

SADT Editor/Diagram Files. The SADT Editor was developed by Steven E, Johnson (8). The SADT diagram and its diagram files produced by the SADT Editor are used for validating the syntax of the SADT diagram. There are two diagram files for the SADT diagram. One file contains the graphical information, and the other file contains the data dictionary information. A summary of the discussion about the SADT Editor is found in Appendix A.

Translator/Translation Rules. The Translator is used to translate the SADT graphical features into formal language descriptions. Several ways to formalize the graphical language have been discussed in Chapter I. Also, requirements for the formalization criteria have been discussed in Chapter II. In this effort, Predicate Logic

is chosen for translating an arbitrary SADT diagram into a set of formulae. This provides proof-theory as a computational definition of notions such as "consistency" and "answer to question". Thus, such a definition will play a main role in the implementation of the SADT Validator. Another reason is that Predicate Logic is easy to understand and represent.

In order to translate an arbitrary SADT diagram into Predicate Logic, the SADT graphical features such as box, arrow, etc. are translated into the Predicates. Some graphical features are mapped into predicates with a one-to- one relationship and some are mapped into the predicates with a many-to-one relationship. Figure 3.3 shows the relationship for mapping the SADT graphical features into the selected predicates.

The graphical features shown in Table 3.1 are the items implemented in the SADT Editor (8:A-5). Since the SADT Validator interfaces with the SADT Editor, the graphical features to be used in the SADT Validator are the same as those of SADT Editor. The graphical feature Box is translated into the predicate BOX(x), which means: x is a BOX. In the case of the ARROW, it is translated into the predicate ARROW(x), which means: x is a ARROW. In the case of INPUT, CONTROL, OUTPUT, and MECHANISM, the graphical features are the attributes of the interface for the arrows connected in a BOX. Therefore, these items are translated into the predicate ATTRIBUTE(x,y,z), which means: z is an attribute of an arrow y for a box x. Thus, the attribute field has one of the values: INPUT, CONTROL, OUTPUT, or MECHANISM. The ACTIVITY NAME is translated into the predicate NAME(x,y), which means : y is a name of box x. The LABELS is translated into the predicate LABEL(x,y), which means: y is a label of an arrow x. In the case of BRANCH, JOIN, BOUNDARY ARROW, 2-WAY ARROW, TUNNEL ARROW, and TO/FROM ALL, these graphical items are translated into the predicate ARROW(x) because each of these is characterized into an arrow. Also, the FOOTNOTE and the SQUIGGLE are translated into the predicate LABEL(x,y) because

Table 3.1. Implemented Graphical Feature and their Predicates

Ross's Article Line Number	Johnson's Term	Predicate
1	BOX	BOX(x)
2	ARROW	ARROW(x)
3	INPUT	ATTRIBUTE(x,y,z)
3	OUTPUT	"
4	CONTROL	"
5	MECHANISM	"
6	ACTIVITY NAME	NAME(x,y)
7	LABELS	LABEL(x,y)
12	BRANCH	ARROW(x)
13	JOIN	"
18	BOUNDARY ARROW	"
22	2-WAY ARROW	"
24	TUNNEL ARROW	"
25	TO/FROM ALL	"
27	FOOTNOTE	LABEL(x,y)
29	SQUIGGLE	"
30	C-NUMBER	-
31	BOX NUMBER	NUMBER
32	MODEL NAME	-
33	ICOM CODE	ICOM(x,y)
37	FACING PAGE TEXT	-

each of these is characterized into a label of an arrow. The BOX NUMBER is translated into NUMBER(x,y) which means: y is a number of a box. Finally, the ICOM CODE is translated into the predicate ICOM(x,y) which means: y is a ICOM code of a arrow x.

Since the SADT Validator interfaces with the SADT Editor, it is also needed to map the data structures of the SADT Editor into the predicates. The discussion of the data structures of the SADT Editor are found in Appendix A. Some data structures are mapped into the predicates with one-to- one relationships, and other structures are mapped into the predicates with many-to-one relationships. Table

Table 3.2. Translation of Data Structure into Predicates

TERM	DATA STRUCTURE NAME	DATA STRUCTURE FIELD	PREDICATES	RELATIONSHIP
BOX	box	struct-type	BOX	one-to-one
ARROW	line	struct-type	ARROW	one-to-one
INPUT	box line	struct-type location struct-type location attribute	ATTRIBUTE	many-to-one
OUTPUT	"	"	"	"
CONTROL	"	"	"	"
MECHANISM	"	"	"	"
ACTIVITY NAME	box	struct-type name	NAME	one-to-one
LABEL	line	struct-type label	ARROW	one-to-one
BRANCH	line	struct-type location attribute	ARROW	one-to-one
JOIN	"	"	"	"
BOUNDARY ARROW	"	"	"	"
2-WAY ARROW	"	"	"	"
TUNNEL ARROW	"	"	"	"
TO/FROM ALL	"	"	"	"
FOOTNOTE	foot- note line	struct-type location label struct-type location	LABEL	many-to-one
SQUIGGLE	squi- ggle line	"	"	"
BOX NUMBER	box number	struct-type	NUMBER	one-to-one
ICOM CODE	line	struct-type location	ICOM	one-to-one

3.2 shows a list of the mapping of the data structures into the predicate with these relationships.

For example, both box and line structures are needed to map the graphic feature INPUT into the predicate ATTRIBUTE(x,y,z). By comparing the location of the box to the location of the line, a decision can be made whether the box contains the line or not. If the line is contained in the box, it can be decided what the attribute of the line is for the box by checking the starting and the ending characteristics of the line. The attribute is one of the values: INPUT, CONTROL, OUTPUT, or MECHANISM.

These mapping relationships between the graphical features (or data structures) and the predicates result in a formation of translation rules. Therefore, an arbitrary SADT diagram can be represented into the formalized forms using the above Predicates through the translation rules.

Syntax Rules. This section presents a list of SADT syntax rules and their representation using the Predicates. It is difficult to define the SADT itself formally, as Ross acknowledged (13:28), because the SA graphical language includes domain related semantics. Thus, the syntax rules implemented in this effort are not complete. However, consistency should be provided among the rules. This work must be a knowledge engineering process. The knowledge of the SADT syntax is represented using the predicates in this thesis effort. A list of SADT syntax rules developed in this thesis effort and their Predicate Logic representations are:

1. Each box must have a name.

   $\forall x, \exists y[BOX(x) \rightarrow NAME(x,y)]$

2. Each box must have a number.

   $\forall x, \exists y[BOX(x) \rightarrow NUMBER(x,y)]$

3. Each arrow must have a label.

$\forall x, \exists y[ARROW(x) \rightarrow LABEL(x, y)]$

4. Each box must have at least one control arrow.

   $\forall x, \exists y[BOX(x) \rightarrow$

   $ARROW(y) \wedge ATTRIBUTE(y, x, 'CONTROL')]$

5. Each box must have at least one output arrow.

   $\forall x, \exists y[BOX(x) \rightarrow$

   $ARROW(y) \wedge ATTRIBUTE(y, x, 'OUTPUT')]$

6. Each diagram has no more than six boxes.

   $\forall x, \exists y[BOX(x) \wedge NUMBER(x, y) \rightarrow$

   $GREATERTHAN(y, 0) \wedge LESSTHAN(y, 7)]$

7. Every box in a diagram must be connected to at least one other box unless there is only one box.

8. External arrows of a diagram should be matched in number and name with the arrows that touch the parent box.

Rule 6 contains two new predicates, which are GREATERTHAN(x,y) and LESSTHAN(x,y). The GREATERTHAN(x,y) predicate implies that x is greater than y, and The LESSTHAN(x,y) predicate implies that x is less than y.

First seven rules are derived with emphasis upon the box and the arrow relationships. The graphical features which present the arrow information such as join, branch, bundle, spread, etc., are not defined in this thesis effort due to the pipeline feature of the SA language. Rule 8 implies the parent and the child relationships. Rule 8 is also not defined because it needs to refer to the two diagrams.

Thus, these syntax rules become a knowledge base of the SADT Validator. The next section discusses the syntax checking process.

Syntax Checker. The main purpose of the syntax checker is to check whether or not an SADT diagram is valid against the SADT syntax rules. When any error is found on the SADT diagram, the appropriate messages are provided in this process. The syntax checker is an inference engine of the knowledge-based system using Predicate Logic representation. Thus, it is needed to discuss how the inference engine works in the knowledge-based system using Predicate Logic representation.

As seen in the previous section, the syntax rules are complex. Thus, these rules can be converted into much simpler forms using the CNF (Conjunctive Normal Form) notation. This conversion process is performed by the following sequence of steps:

1. Eliminate the implication $\rightarrow$, using the fact that a $\rightarrow$ b is equivalent to $\sim$a $\bigvee$ b.
2. Reduce the scope of $\sim$, using the fact that $\sim (\sim p) = p$, deMorgran's laws and the standard correspondences between quantifiers [$\sim \forall x P(x) = x \sim \exists P(x)]and[\sim \exists x P(x) = x \sim \forall P(x)]$.
3. Standardize variables so that each quantifier binds a unique variables. For example, the formula x P(x) $\bigvee$ x Q(x) would be converted to x P(x) $\bigvee$ y Q(y).
4. Move all quantifiers to the left of the formula.
5. Eliminate existential quantifiers $\exists$ with appropriate substitution of Skolem constants and function.
6. Drop the universal quantifiers $\forall$.
7. Convert to conjunctive normal forms.
8. Eliminate conjunctions so that conjunctive normal forms can be formed into a list of clauses.
9. Rename variables so all clauses are unique [11:151-152].

After applying this entire procedure to a set of Predicate Logic representations from an SADT diagram and syntax rules, a set of clauses will be produced, each of which is a disjunction of literals. These clauses can now be exploited by the resolution procedure to generate the output messages by the syntax checker.

> The resolution procedure is an iterative process, at each step where two clauses, called the parent clauses, are compared. The result yields a new clause called resolvent. This resolution procedure needs a matching procedure that compares two clauses and discovers whether there exists a set of substitutions that makes them identical. This matching procedure is called the unification algorithm [11:157].

The general resolution procedure for Predicate Logic is performed by the following steps in sequence:

1. Convert the SADT syntax rules represented by Predicate Logic into CNF (Conjunctive Normal Form). The result yields a set of clause forms.
2. Negate an SADT diagram represented by Predicate Logic to be proved, and convert the result to clause form. Add it to the set of clauses obtained in 1.
3. Repeat until either a contradiction is found, no progress can be made, or a predetermined amount of effort has been expended:
   (a) Select two clauses. Call these the parent clauses.
   (b) Resolve them together. The resolvent will be the disjunctive of all of the literals of both of the parent clauses with appropriate substitutions performed.
   (c) If the resolvent is the empty clause, then a contraction has been found. If it is not, then add it to the set of clauses available to the procedure [11:158].

However, during the resolution procedure, if no contradiction exists, it is possible the resolution procedure will never terminate. This is a completeness problem. A way of detecting that no contradiction exists is required.

> But, from a computational point of view, completeness is not an important question. Instead, we are much more interested in whether good enough heuristics can be discovered so that a proof can be found in the limited amount time that is available [11:168].

Also, when this general procedure is applied to the syntax checker, the result yields either true or false. In other words, if there is any error on the SADT diagram, then the resolution procedure yields false. However, in order to provide good user interface, it is necessary to explain why the error is produced. Detailed implementation about the explanation facility is presented in Chapter IV.

Summary

This chapter presented a conceptual design decisions for the SADT Validator. The requirement analysis dictated design of two main functions, which were the translator and the syntax checker. Thus, design decisions for these functions were discussed. Also, Predicate Logic was chosen to represent the SADT graphical features and the syntax rules. The interface of this representation with the SADT Editor was also addressed. The next chapter presents detailed design, implementation, and test of the SADT Validator.

IV. Detailed Design, Implementation, and Test

Introduction

This chapter presents a detailed design of the components specified in the conceptual design chapter. The major components identified were the translator and the syntax checker. Within the syntax checker, two sub- components were identified. The first was an inference engine or control procedure. The second component was the knowledge base which consists of the syntax rules and the domain information of the SA graphical features. Also, this chapter presents the implementation issues of the SADT Validator, and reviews the testing approach used during the development.

Detailed Design of Translator

The translator is used to translate an SADT diagram into the Predicate Logic forms. The mapping relationships between the SA graphical features and their predicates were presented in the conceptual design chapter.

The data structure of the translator is designed with emphasis upon the box because the box of an SADT diagram plays the most important role in the presentation of the activity models. The box structure consists of three fields: box name, box number, and connecting arrows. Each arrow in the box structure consists of three fields: arrow name, attribute, and ICOM code. The attribute of the arrow identifies one of the following values: input, control, output, or mechanism. Thus, each box in an SADT diagram has one box structure.

A discussion is necessary to describe the process through which the data structure can be obtained from an SADT diagram. This issue needs some clarification regarding the box structure and the data structures of the SADT Editor. The data structures of the SADT Editor are discussed in Appendix A.

The box name and number of the box structure can be directly matched with their fields in the box structure of the SADT Editor. However, connecting arrows are not simple. By comparing the location of the box to the location of the line from the box and the line structure of the SADT Editor, a decision can be made whether or not the box contains the line. If the line is contained in the box, the arrow name and the ICOM code can be directly obtained from the line structure. Also, it can be determined what the attribute of the line is by checking the starting and the ending attributes of the line. The attribute of the arrow must be one of the following values: input, control, output, or mechanism.

Detailed Design of Syntax Checker

The syntax checker is used to check the syntax of the SA graphical features for an SADT diagram, and to produce the appropriate result message. Within the syntax checker, two sub-components were identified: the knowledge base and the inference engine. The knowledge base consists of the syntax rules and the facts. The facts are the Predicate Logic forms produced from an SADT diagram by the translator.

Knowledge Base. The knowledge base for the syntax checker consists of two sub-components. The first includes the syntax rules of the SADT method. The second consists of the facts, which are the predicate forms produced from an SADT diagram.

If-then rules are chosen for the representation of the syntax rules because these usually turn out to be a natural form of expressing knowledge. Also, if-then rules have the following additional desirable features:

1. Modularity: each rule defines a small, relatively independent piece of knowledge.

2. Incrementability: new rules can be added to the knowledge base relatively independently of other rules.
3. Modifiability (as a consequence of modularity): old rules can be changed relatively independently of other rules.
4. Support system's transparency [4:316-317].

For example, the syntax rule, which is "each box must have a number", is represented by the following if-then rule:

- Rule: if there is a box
- and the box does not have a number
- then there is a number error on the box.

If-then rules for the syntax rules are presented in Appendix D.

The "if" part of the if-then rule is the condition and the "then" part is the conclusion. Thus, the facts should be matched with the condition part. Also, the conclusion part presents either the new fact or an explanation of an error condition. In addition, the knowledge base includes the relation "askable" which defines those things that can be asked of the user.

Inference Engine. An inference engine determines the appropriate use of the knowledge in the knowledge base. This inference engine includes an interface between the user and the system. Thus, the inference engine provides the user with an insight into the problem-solving process carried out by the inference engine. The reasoning process of the inference engine is performed according to the following principles:

To find an answer Answ to a question Q use one of the following:

1. If Q is found as a fact in the knowledge base, then Answ is 'Q' is true.

2. If there is a rule in the knowledge base of the form 'if Condition then Q', then explore Condition in order to find answer Answ.
3. If Q is an 'askable' question, then ask the user about Q.
4. If Q is of the form Q1 and Q2, then explore Q1 and now: if Q1 is false, then Answ is 'Q is false', else explore Q2 and appropriately combine answers to both Q1 and Q2 into Answ.
5. If Q is of the form Q1 or Q2, then explore Q1 and now: if Q1 is true, then Answ is 'Q is true', or alternatively explore Q2 and appropriately combine answers to both Q1 and Q2 into Answ [4:326-327].

Also, this inference engine should include the process to trace how the conclusion was reached for the user's understanding.

Implementation

Language Issues. For the translator, a decision was made to proceed using C because the SADT Editor was implemented using C. Thus, many portions of the SADT Editor could be reused without modifications.

During the implementation of the syntax checker, numerous problems were encountered while using Xenologic Prolog on the Sun workstation. For example, when the Xenologic Prolog was used abnormally, the Sun workstation became inoperative. Thus, the user was forced to re-boot the system. Another example was that, when the program using this language had many syntax errors, the Sun workstation became inoperative. Thus, Xenologic Prolog was not proper for this effort. However, the MS-DOS Prolog-1 Version 2.2, which was developed by Expert System Ltd., was available for the Z-248 workstation. Therefore, the syntax checker was implemented on the Z-248 workstations.

Interfaces. Since Prolog-1 was chosen for the implementation of the syntax checker, the predicate forms, which are the output of the translator, should be stored in a file. A list of rules is presented in Appendix C and a list of facts is presented in Appendix D. As a result, the file for the predicate forms became an input file for the syntax checker. Thus, the file included the facts of the knowledge base for the syntax checker.

Implementation of Syntax Checker. Two sub-components were identified within the syntax checker in the design section. An inference engine, called BC3, which was a shell for backward-chaining expert system, was available. Thus, BC3 was used for the inference engine of the syntax checker.

In order to use BC3 as the inference engine for the syntax checker, each item of the facts was represented by a triple, a three-element list of the form: [Object, Attribute, Value]. In addition, an associated knowledge base supplies the following data:

1. A goal statement, in the form of a list of triples to be solved in sequence. The solved triples are printed.
2. A collection of if-then rules for triples.
3. A collection of fact triples.
4. A collection of 'askable' triples, indicating the forms of triples whose values may be obtained from the user.
5. A collection of 'keep' triples, indicating the form of the triples not to be erased from working memory at the beginning of a new session.

When each syntax rule is applied to the facts, a message is produced by a goal statement. Also, each syntax rule is represented using the if-then rule. In addition, each of the predicate forms is represented into the fact using a three- element list of

form: [Object, Attribute, Value]. The knowledge base of the syntax checker has one 'askable' triple, which is a box name to be checked.

Format of Predicate File. This sub-section presents the format of the predicate file. The file includes the predicate forms of an SADT diagram produced by the translator. Since each item of the file is used as the fact of the knowledge base for the syntax checker, it should be represented using a three-element list of the form: [Object, Attribute, Value]. An example of the contents of the file is presented in Appendix D.

Documentation Standard

Internal documentation of the code follows that prescribes in the *AFIT/ENG Software Development Documentation Guidelines and Standards.* Each program file begins with the standard file header (7:38) and each module also begins with the standard header (7:40). In addition, C and Prolog-1 language comments are provided in the code to amplify and clarify each section of the code.

Test

The testing approach used in developing the SADT Validator occurs in four phases. These phases are unit testing, integration testing, validation testing, and system testing (10:502). These phases are shown in Figure 4.1.

Unit testing examines a module's interface, data structure integrity, boundary conditions, and error handling (10:503-504). Each of these areas is tested using both test data and normal usage of the modules. Because of the extensive data passing between modules, the module's ability to maintain a structure's integrity is emphasized.

Integration testing focuses on uncovering interface errors(10:507). A bottom-up incremental integration test is used (10:508). This method was selected because

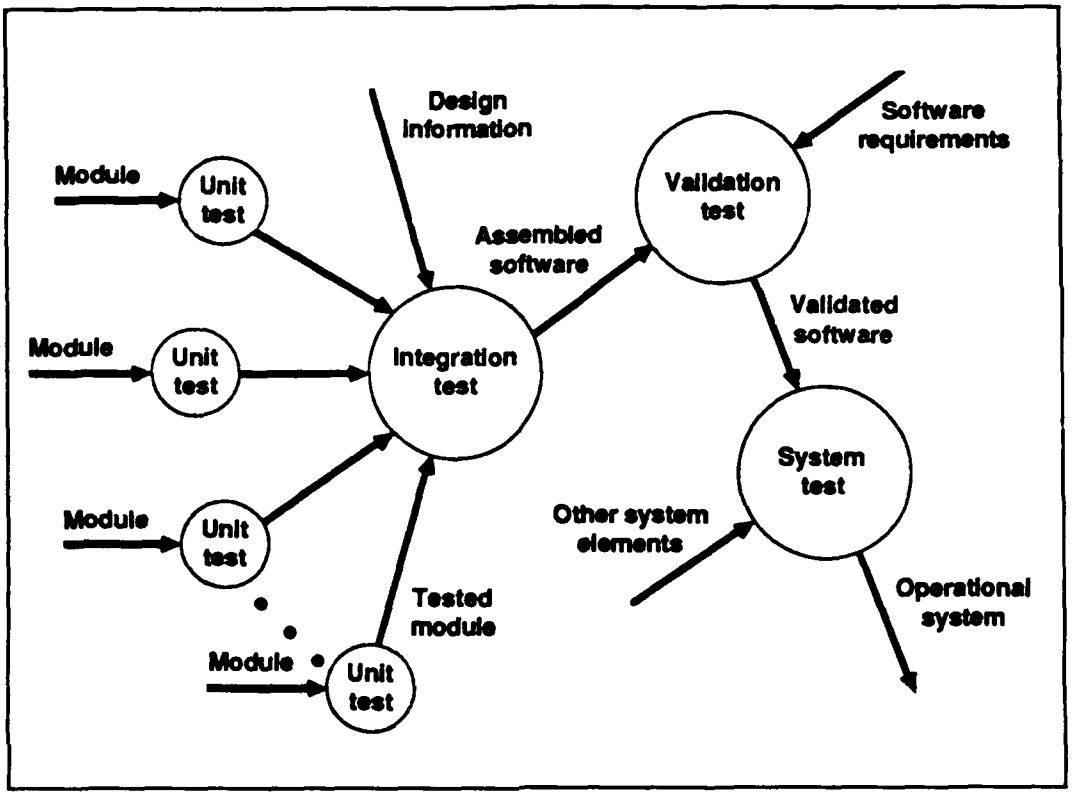


Figure 4.1. Software Testing Steps (10:503)

low level modules of the SADT Validator contains the retrieval routines for the translation and the syntax rule base.

Validation testing occurs next. This testing phase is concerned with the "does it work as expected" question (10:514). This method is used to check whether or not the SADT Validator finds errors on the SADT diagram consistently as expected.

System testing is concerned with overall system issues, such as software and hardware compatibility, and usually involves different test groups (10:516).

Summary

This chapter discussed the detail design decisions based on the conceptual design and the objectives of the thesis. For the translator, C was chosen because of portability of the tool and reusability of the Johnson's code. For the syntax checker, MS-DOS Prolog-1 was chosen because the Xenologic Prolog had many problems. Thus, the translator was implemented on the Sun workstation using C. The syntax checker was implemented on the Z-248 workstation using Prolog-1. Also, coding and testing approaches were presented in the last two parts of the chapter. The next chapter presents the conclusions from conducting this thesis and recommends several future studies as a result of this effort.

V. *Conclusions and Recommendations*

Conclusions

The objective of this thesis effort was to perform the prototype development of a syntax validation tool for graphical diagrams using SADT methodology.

This effort was performed in three phases. During the first phase, the formal definition of the SADT methodology was derived with emphasis upon the syntax of the SA graphical features using Predicate Logic representation. The formal syntax definition of the SADT methodology derived in this effort was not complete but consistent because the SA graphical language also includes semantics, as Ross acknowledged (13:28). During the second phase, the SADT Editor was analyzed and the interface issues with the associated software were identified. Thus, the graphical items were translated into their predicates. The predicates were derived with emphasis upon the box and the arrow relationship of the SADT diagram. During the third phase, the syntax rules of the SADT methodology were derived using Predicate Logic representation. The syntax rules were also identified with emphasis upon the box and the arrow relationships.

The syntax checking process was implemented using a knowledge based system to ease the extension of the syntax rules, to add knowledge of the SADT semantics, and to add domain knowledge of the application system developed by the SADT methodology. Unfortunately, Xenologic Prolog on the Sun workstation was unstable during the implementation. Thus, the syntax checking part of the SADT Validator was implemented on the Z-248 workstation using Prolog-1. The translation part was implemented as an integral part of the SADT Editor on the Sun workstation using the graphic package SunView and C under Sunwindow environment.

This thesis effort was successfully accomplished and a prototype syntax validation tool of the SADT methodology was designed and developed. However, several

aspects of the tool were identified that could be enhanced. These aspects are presented in the next section.

Recommendations

This section presents recommendations for future studies which could lead to further improvements in the tool:

1. Extend the formal definition of the SADT syntax. This issue needs to analyze the SA graphical features in more detail. Currently, the tool has only the formal definition with emphasis upon the box and the arrow relationships. For example, the graphical features which present arrow information such as join, branch, bundle, spread etc. should be identified with formal meanings. Also, the relationships between parent diagram and child diagram should be defined to improve the capability of the tool.
2. Add more syntax rules. This issue is dependent upon the extension of the formal definition of the SADT syntax.
3. Integrate the translation process with the syntax checking process. This issue needs to address the stability of the Xenologic Prolog. Also, the Xenologic Prolog should have an interface with C. Although this interface was referenced in the Xenologic Prolog manual (22:8- 29), it was inoperative on the Sun workstation.
4. Apply the knowledge base of the syntax checker to the design knowledge of the application software system. The design knowledge using the SADT methodology can be reused for new software development in a similar design. This project needs the development of a design schema which represents the knowledge of the software design component. Design schemas provide a means for abstracting software designs into broadly reusable components that can be assembled and refined into new software designs. Thus, the system's knowl-

edge base uses SADT model segments to represent design components. These segments are combined and refined, using transformation rules, to produce a SADT model of a design that leads to the goal program. Thus, the system's knowledge base includes design schemas for SADT model segments and their domain information. The main application of the design schema is the design reusability in the software development. The existing software components could be used for new software design in a similar domain. Thus, the labor and the time required in the development of a new software could be reduced. Whenever new software is developed, its design components are stored into the system's knowledge base using the design schema representation with its domain information for later use.

Summary

This chapter presented the conclusions drawn from the design and development of a syntax validation tool. Additionally, the recommendations for further research were identified.

Appendix A. *Summary of SADT Editor*

Introduction

The purpose of this appendix is to summarize the discussion of the SADT Editor developed by Steven E. Johnson (8). The SADT Editor has been developed for extending the previous SADT Editor, which had been developed by James W Urscheler, by providing a more complete set of the SADT graphical features (8) (21).

A discussion of Johnson's SADT Editor is necessary because the SADT Validator should interface with the SADT Editor and use an SADT diagram and its diagram files produced from the SADT Editor. The SADT Editor allows users to interactively create and edit structured analysis diagrams. In addition, partial data dictionary information is automatically generated from graphics information by the SADT Editor and is supplemented by inputs from the user through the SADT Editor.

Implemented Graphic Features

Figure A.1 shows the graphic features implemented in the SADT Editor. Column 1 in the figure presents the line numbers of the whole graphic features developed by Ross (14:20). Column 2 presents the terms of the graphic features. Column 3 presents the page of the User's Manual Reference in Johnson's thesis. All graphic features developed by Ross were not implemented due to the time limitation.

Screen Layout

Figure A.2 shows the screen layout used in the SADT Editor. There are five windows: the Input Window, the Message Window, the Selection Window, the Diagram Window, and the Data Dictionary Window.

Ross article line number	Term	User's Manual Reference
1	BOX	2-2,3
2	ARROW	2-2,3
3	INPUT	3-26 (FIG)
3	OUTPUT	3-26 (FIG)
4	CONTROL	3-26 (FIG)
5	MECHANISM	3-11
6	ACTIVITY NAME	2-3,4
7	LABEL	2-3,4
12	BRANCH	3-9
13	JOIN	3-9
18	BOUNDARY ARROW	3-17
22	2-WAY ARROW	3-26 (FIG)
24	TUNNEL ARROW	2-3
25	TO/FROM ALL	6-21
27	FOOTNOTE	3-26 (FIG)
29	SQUIGGLE	3-26 (FIG)
30	C-NUMBER	2-3
31	NODE NUMBER	2-3
32	MODEL NAME	3-16
33	ICOM CODE	4-8
37	FACING PAGE TEXT	4-1

Figure A.1. Implemented Graphic Syntax (8:A-5)

The Input Window, which is located at the top of the screen, is used for displaying the keyboard input. Errors are correctable using the DELETE key on the keyboard, and input typed is effected by using the RETURN key.

The Message Window, just under the Input Window, is used to help and respond to user's operations. Thus, the current status of the tool is shown in the Message Window.

The Selection Window, just under the Message Window, is used to select the menu which users desire to operate. There are five ovals on the Selection Window:

RECALL DIAGRAM, EDIT DD, EDIT FPT, MISC FUNCTIONS, and SAVE DIAGRAM.

The RECALL DIAGRAM oval is used to read an existing diagram file. The EDIT DIAGRAM oval is used to create and edit a diagram. The EDIT DD oval is used to create and edit data dictionary information. The EDIT FPT oval is used for facing page text of a diagram. The MISC FUNCTIONS oval is used for miscellaneous functions such as, making a dump file for a diagram, exiting the SADT TOOL, etc.. The SAVE DIAGRAM is used to save graphics files and data dictionary files of the current diagram.

The Diagram Window, just under the Selection Window, is used to draw all graphics features, and to display text typed in the Input Window.

The Data Dictionary Window is used to edit the data dictionary information, which cannot be accessed from its diagram.

Finally, the SADT Editor uses a menu-driven system to provide good user interface. Figure A.3 shows all the menu selections provided by the SADT Editor and their hierachical structure. These menu items are selected in the Selection Window on the screen layout using the mouse button. Detailed description is found in the User's Manual of Johnson's thesis (8).

Data Structure

There are five structures: the box structure, the line structure, the squiggle line structure, the header structure, and the footnote structure (8:4-11 - 4-14).

The box structure contains the information about the location, the label, and the numeric structure type (8:4-11). "All the activity boxes on the diagram are maintained by using a linked list. Also, each box structure uses a C pointer to point to activity data dictionary structure (8:4-11)".

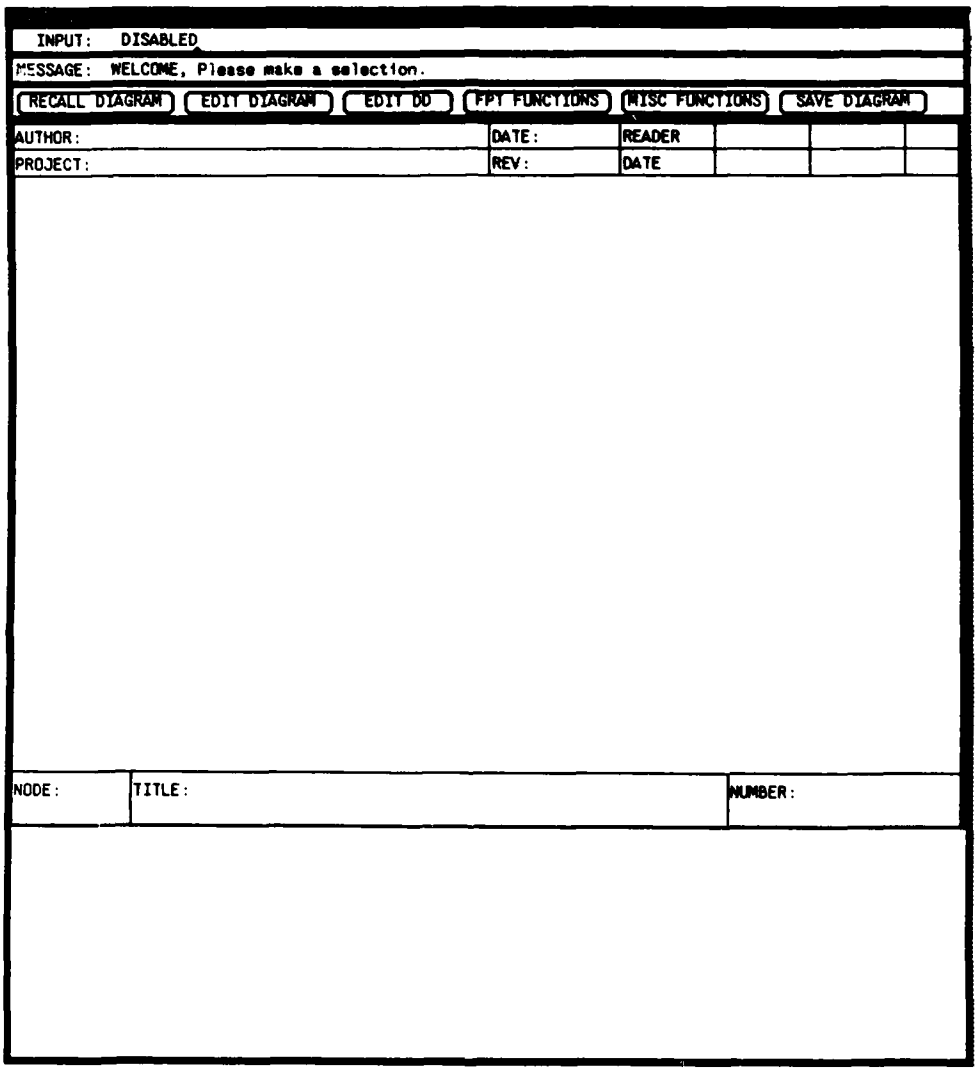


Figure A.2. Screen Layout of SADT Editor

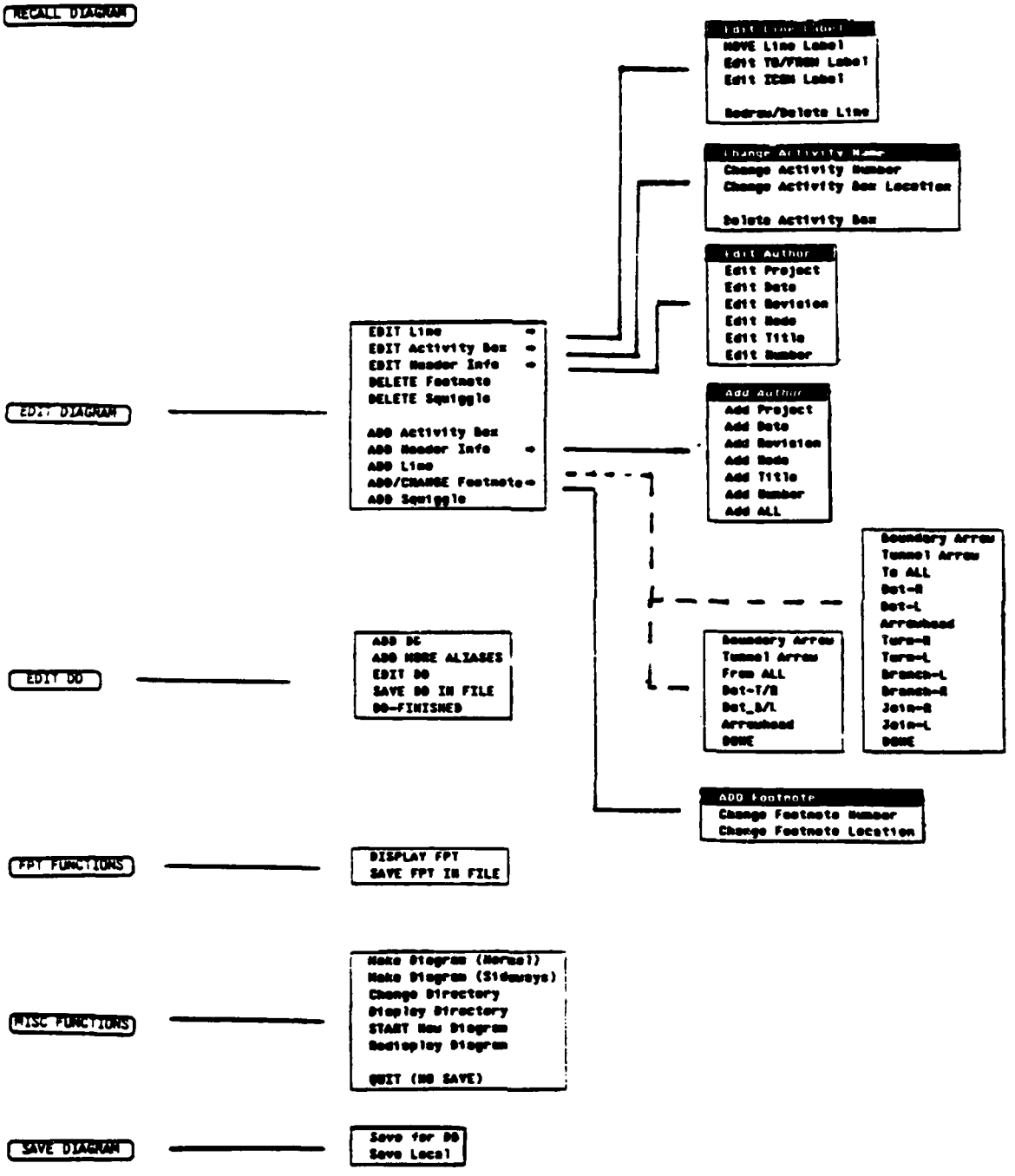


Figure A.3. SADT Editor Menus

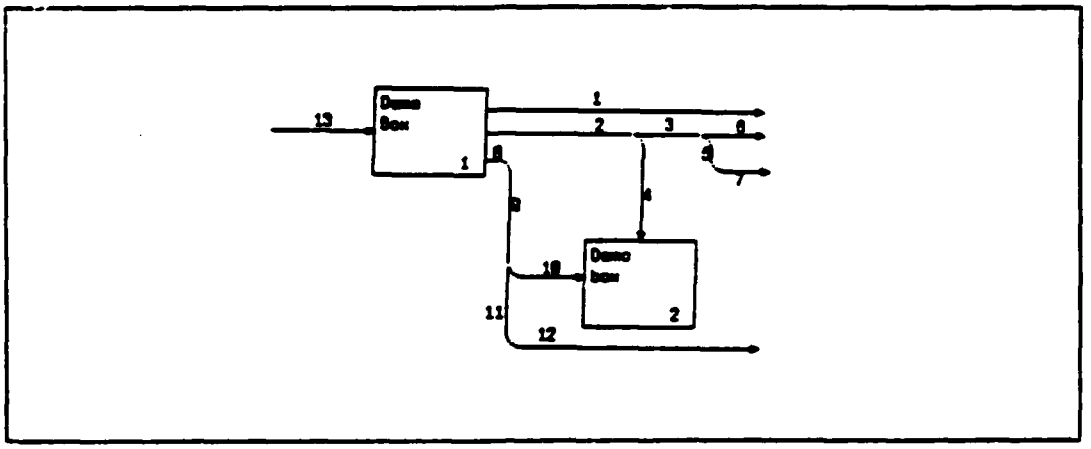


Figure A.4. Example Group of Lines (8:4-12)

The line structure contains the information about the location, the label, the numeric structure type, the ICOM field, and the TO/FROM ALL field (8:4-11 - 4-12). In addition, "two numbers are defined to identify the graphical entities drawn on each end of the line (ie. arrowhead, tunnel, dot, turn right, or branch left, etc.). Finally, the lines are stored in binary trees with the root nodes linked to other root nodes by C pointers (8:4-12)". For example, Figure A.4 shows three groups of lines and the corresponding linked list structure is shown in Figure A.5 (6:4-12).

Figure A.5 shows that the tree arrangement is advantageous in depicting how the line segments actually connect to one another (8:4-12). Also, C supports the simple recursive functions used for traversing binary trees.

Also, "each line structure uses a C pointer to point to a data dictionary structure representing a data dictionary for a data element (8:4-13)".

The squiggle line structure contains the information about the location and the numeric structure type field (8:4-13). Also, "the squiggle lines for a particular diagram are stored in a singly linked list; therefore, each structure contains a C pointer to another squiggle line structure (8:4-13 - 4-14)".

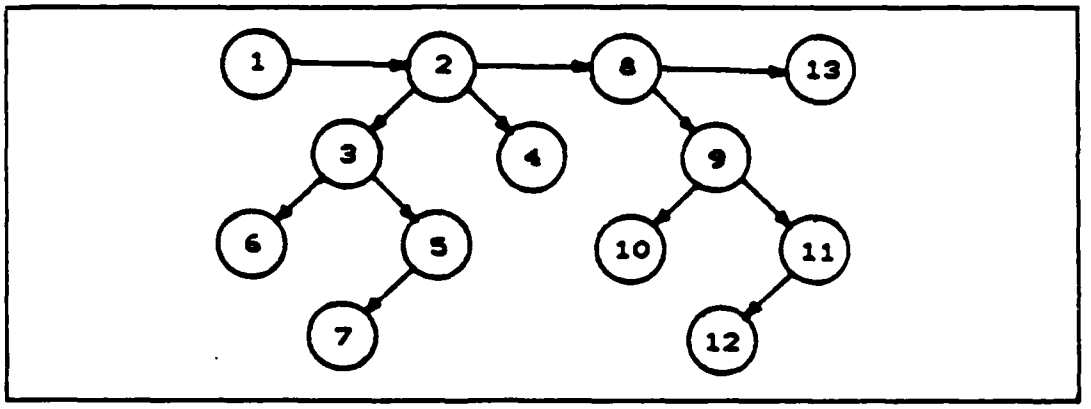


Figure A.5. Resulting Linked List (8:4-13)

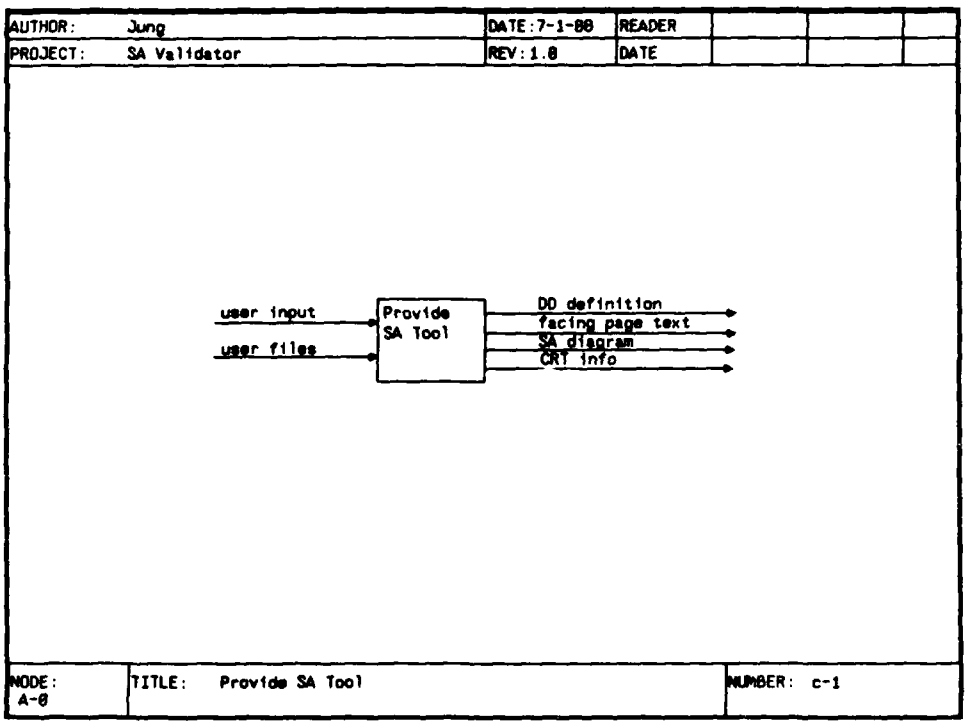
The header structure consists of seven fields: AUTHOR, DATE, PROJECT, REV, NODE, TITLE, and NUMBER (8:4-14 - 4-15).

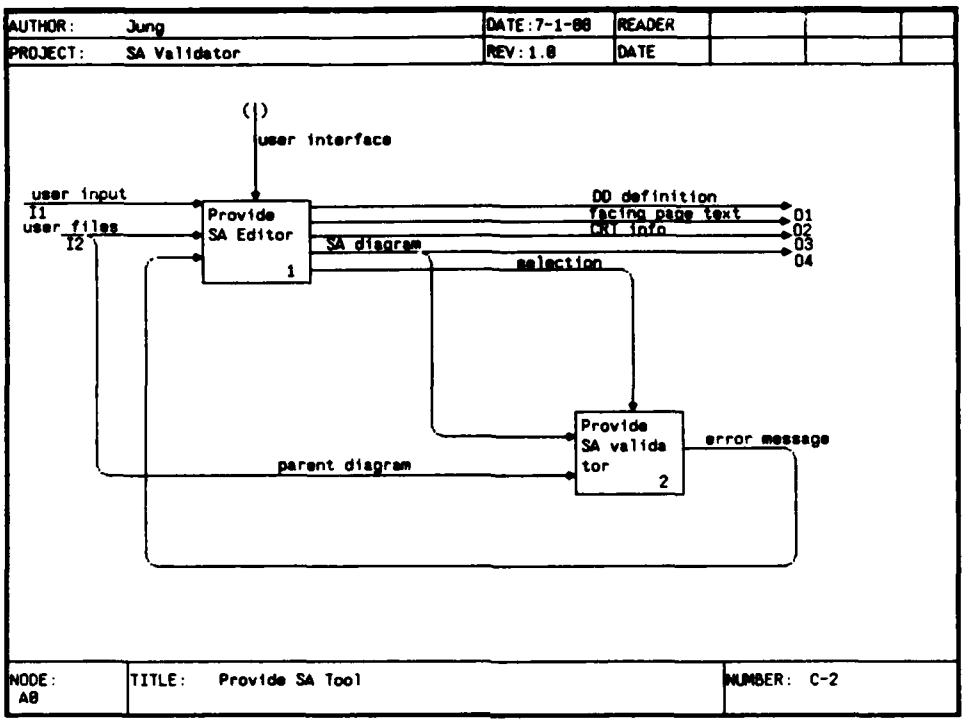
Finally, the footnote structure contains "all the information needed to draw, locate and classify a matching pair of footnote labels (8:4-14)". Also, "the footnote structures for a diagram are stored in a singly linked list; therefore, a C pointer to another footnote structure is defined (8:4-14)".

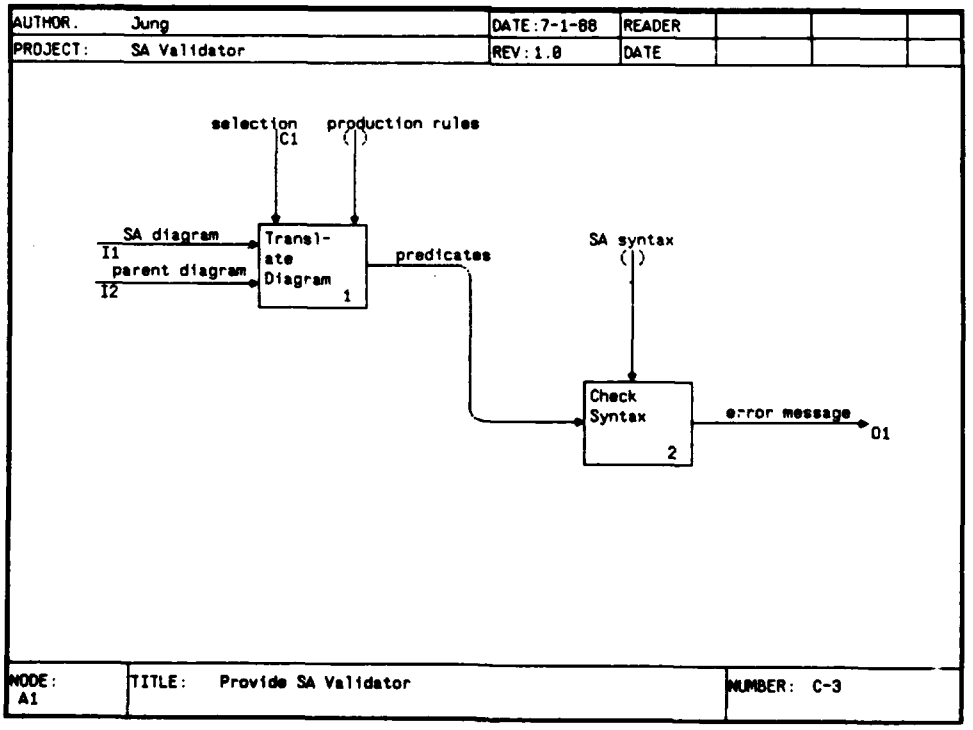
Summary

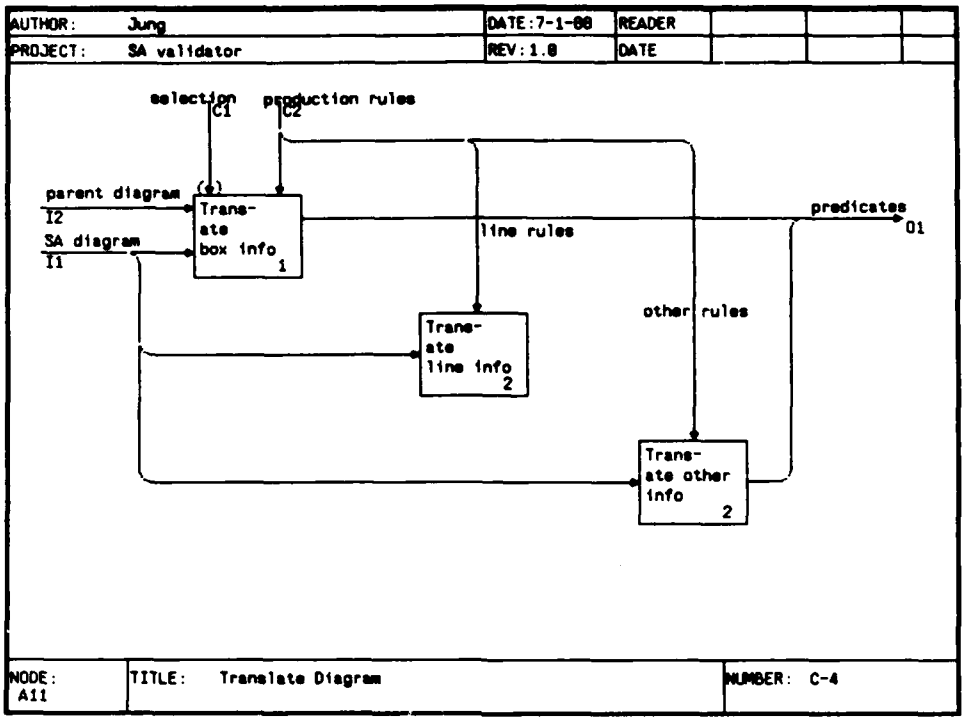
In this appendix, the information needed in the development of the SADT Validator was presented from Johnson's thesis. This information was used throughout the requirement analysis, the design, and the implementation of this thesis effort.

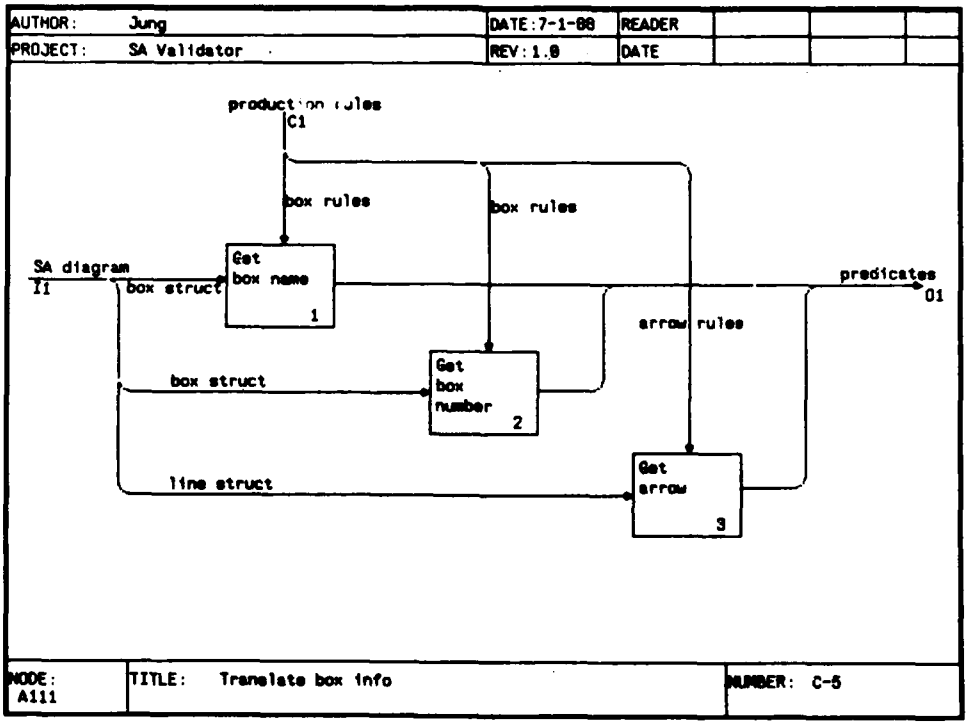
Appendix B. *Requirements Analysis Diagram*

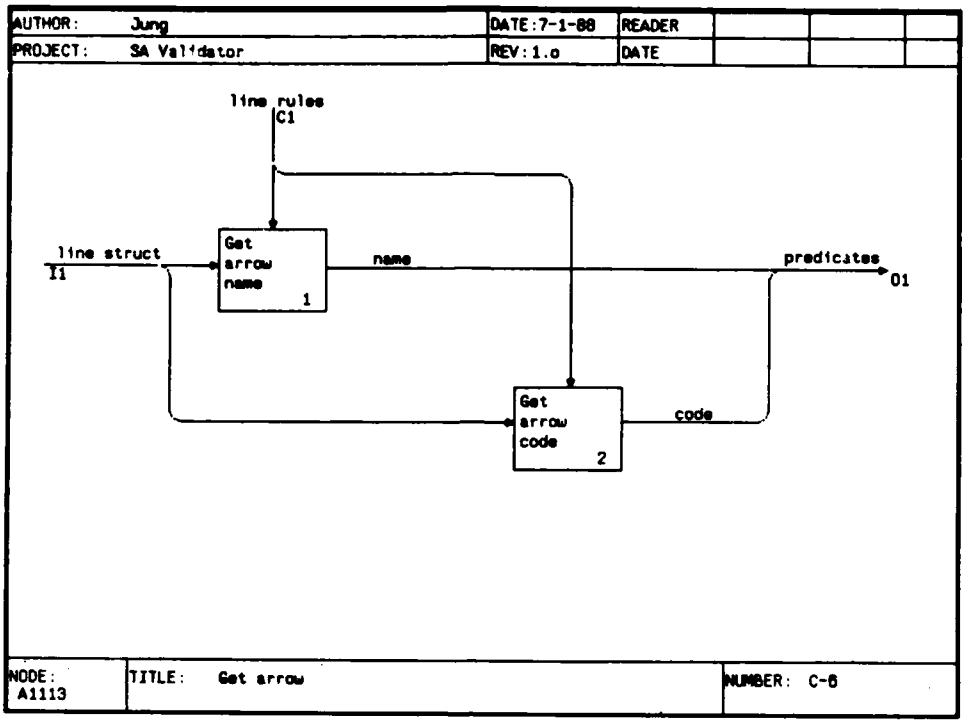


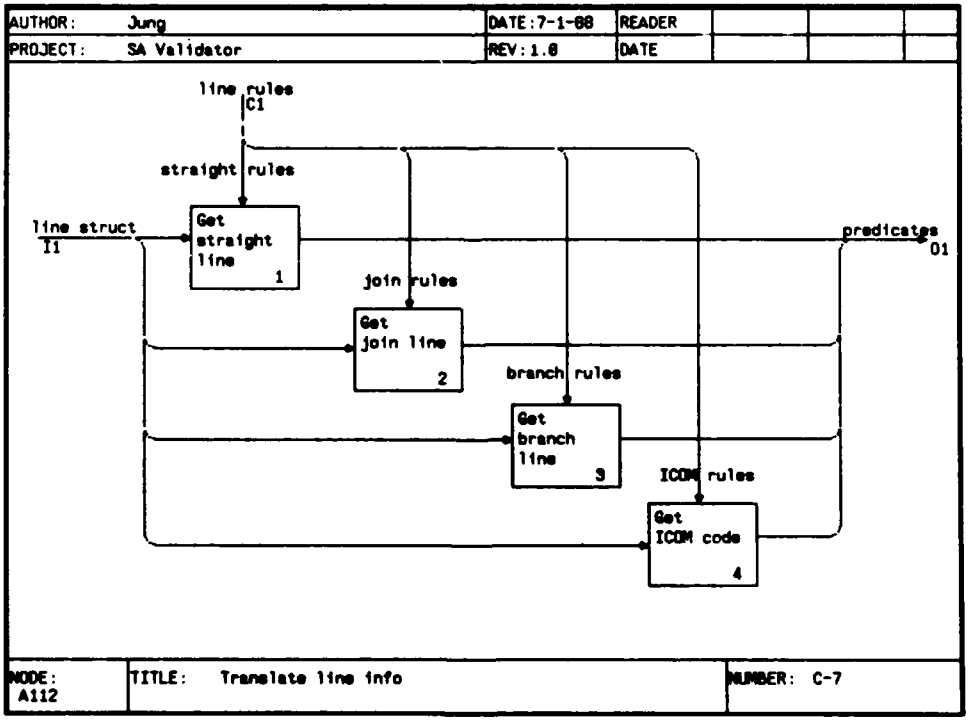

AUTHOR: Jung
DATE: 7-1-88
READER
PROJECT: SA Validator
REV: 1.0
DATE
user interface
user input
I1
user files
I2
Provide SA Editor
1
DD definition
facing page text
CRT info
SA diagram
selection
01
02
03
04
Provide SA valida tor
2
error message
parent diagram
NODE: A0
TITLE: Provide SA Tool
NUMBER: C-2

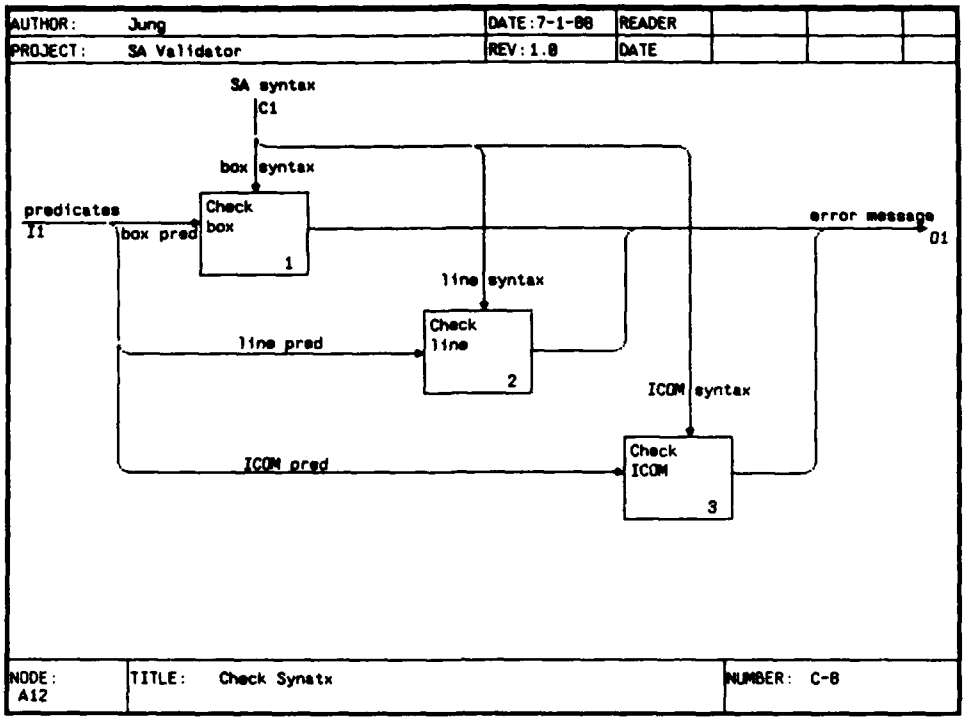

AUTHOR: Jung
DATE: 7-1-88
READER
PROJECT: SA Validator
REV: 1.0
DATE
selection
C1
production rules
SA diagram
I1
parent diagram
I2
Transl- ate Diagram
1
predicates
SA syntax
Check Syntax
2
error message
01
NODE: A1
TITLE: Provide SA Validator
NUMBER: C-3

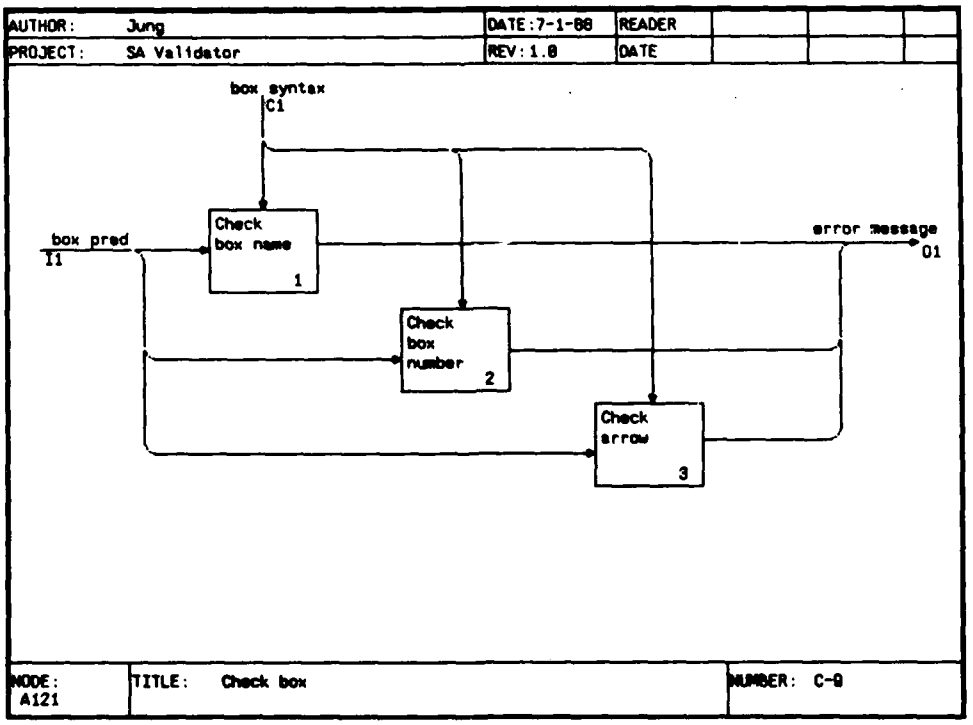

AUTHOR: Jung
DATE: 7-1-88
READER
PROJECT: SA validator
REV: 1.0
DATE
selection
C1
production rules
C2
parent diagram
I2
SA diagram
I1
Trans-
ate
box info
1
line rules
predicates
O1
other rules
Trans-
ate
line info
2
Trans-
ate other
info
2
NODE:
A11
TITLE: Translate Diagram
NUMBER: C-4

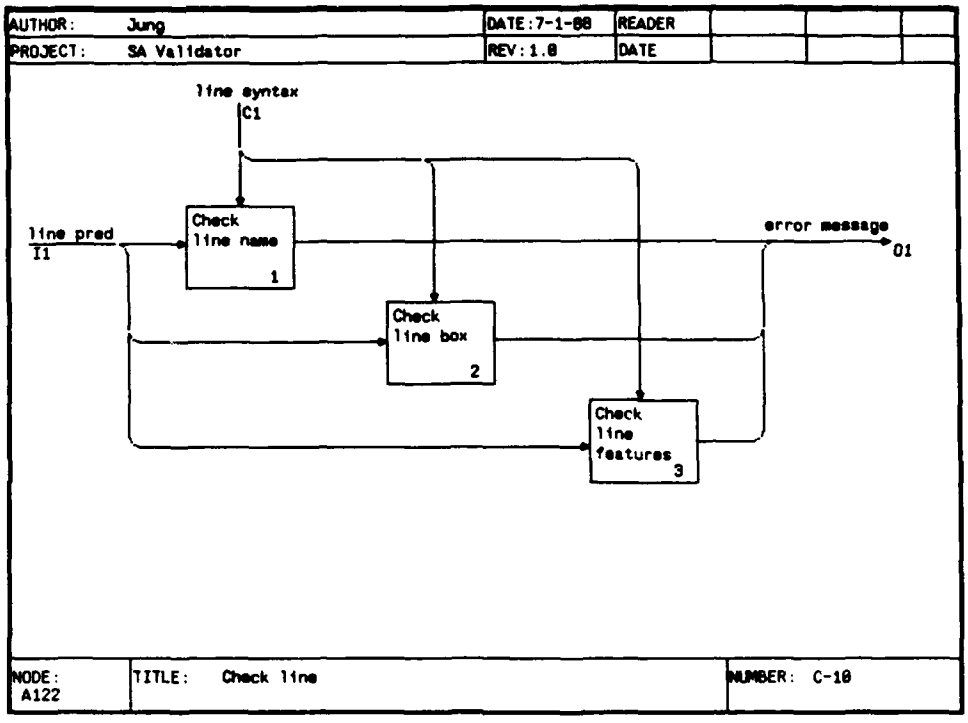

AUTHOR: Jung
DATE: 7-1-88
READER
PROJECT: SA Validator
REV: 1.0
DATE
production rules
C1
box rules
box rules
SA diagram
I1
box struct
Get
box name
1
predicates
O1
arrow rules
box struct
Get
box
number
2
line struct
Get
arrow
3
NODE:
A111
TITLE: Translate box info
NUMBER: C-5

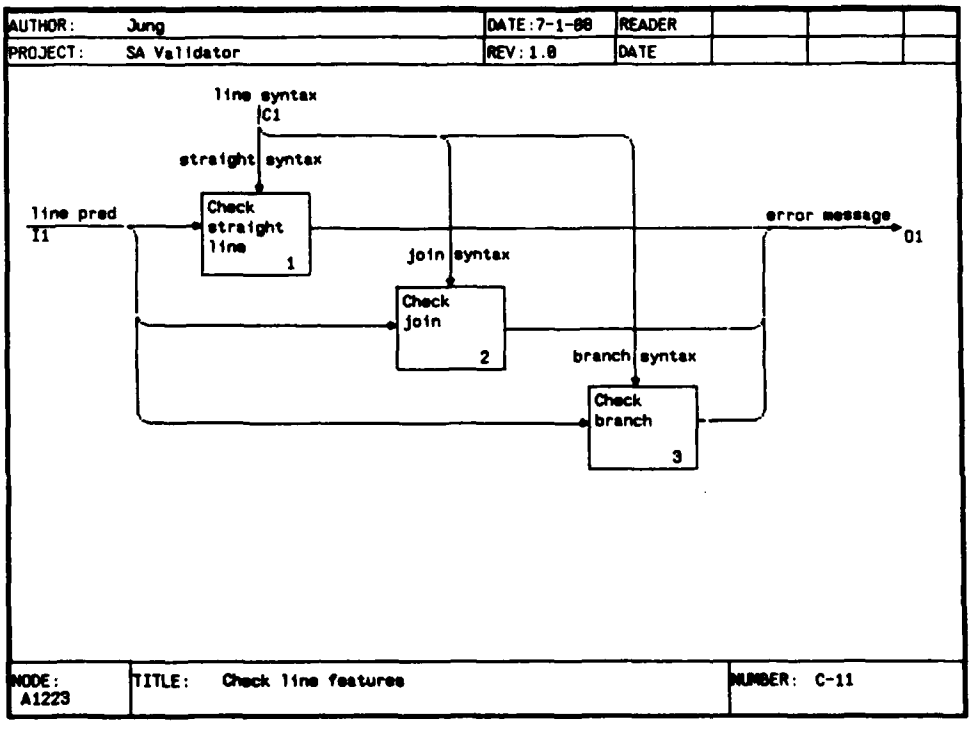

AUTHOR: Jung
DATE:7-1-88
READER
PROJECT: SA Validator
REV:1.0
DATE
line rules
C1
line struct
I1
Get
arrow
name
1
name
predicates
01
Get
arrow
code
2
code
NODE:
A1113
TITLE: Get arrow
NUMBER: C-6


AUTHOR: Jung
DATE:7-1-88
READER
PROJECT: SA Validator
REV:1.0
DATE
line rules
C1
straight rules
line struct
I1
Get
straight
line
1
predicates
01
join rules
Get
join line
2
branch rules
Get
branch
line
3
ICOM rules
Get
ICOM code
4
NODE:
A112
TITLE: Translate line info
NUMBER: C-7


AUTHOR: Jung
DATE:7-1-88
READER
PROJECT: SA Validator
REV:1.0
DATE
SA syntax
C1
box syntax
predicates
I1
box pred
Check box
1
error message
O1
line syntax
Check line
2
line pred
ICOM syntax
Check ICOM
3
ICOM pred
NODE: A12
TITLE: Check Syntax
NUMBER: C-8


AUTHOR: Jung
DATE:7-1-88
READER
PROJECT: SA Validator
REV:1.0
DATE
box syntax
C1
box pred
I1
Check box name
1
error message
O1
Check box number
2
Check arrow
3
NODE: A121
TITLE: Check box
NUMBER: C-9


AUTHOR: Jung
DATE: 7-1-88
READER
PROJECT: SA Validator
REV: 1.0
DATE
line syntax
C1
line pred
I1
Check
line name
1
error message
O1
Check
line box
2
Check
line
features
3
NODE:
A122
TITLE: Check line
NUMBER: C-10


AUTHOR: Jung
DATE: 7-1-88
READER
PROJECT: SA Validator
REV: 1.0
DATE
line syntax
C1
straight syntax
line pred
I1
Check
straight
line
1
error message
O1
join syntax
Check
join
2
branch syntax
Check
branch
3
NODE:
A1223
TITLE: Check line features
NUMBER: C-11

Appendix C. *Source Code List*

Translator

```
/*****************************************************************
*                                                                *
*   DATE:    2  July 1988                                        *
*   VERSION: 1.0                                                 *
*                                                                *
*   NAME:    translate.c                                         *
*   DESCRIPTION:                                                 *
*   This file contains the functions needed to translate all     *
*   graphical features from the SA diagram into their            *
*   predicates.                                                  *
*                                                                *
*                                                                *
*   OPERATING SYSTEM:  UNIX                                      *
*   LANGUAGE:          C                                         *
*   CONTENTS:        *                                           *
*                                                                *
*                  store_diagram()                               *
*                  search_box()                                  *
*                  init_translate()                              *
*                  translate_diagram()                           *
*                  check_syntax()                                *
*   AUTHOR:  Donghak Jung                                        *
*   HISTORY:                                                     *
*****************************************************************/



#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>
#include <suntool/panel.h>
#include <suntool/textsw.h>
#include <sys/param.h>
#include "globals.h"
```

```
/*****************************************************************
*                                                                *
*   DATE:    1  July 1988                                        *
*   VERSION: 1.0                                                 *
*                                                                *
*   NAME:            store_diagram()                             *
*   MODULE NUMBER:                                               *
*   DESCRIPTION:                                                 *
*   This purpose of this module is to store all the predicates*
*   of a given diagram into a file.  (.pro  extension)           *
*   ALGORITHM:                                                   *
*   PASSED VARIABLES:        fp,bbox                             *
*                                                                *
*   RETURNS:                                                     *
*   GLOBAL VARIABLES USED:  header_rootnode,box                  *
*   GLOVAL VARIABLES CHANGED:   box                              *
*   FILES READ                                                   *
*   FILES WRITTEN:  file pointer passed to a .pro file           *
*   HARDWARE INPUT:                                              *
*   HARDWARE OUTPUT:                                             *
*   MODULES CALLED: itoa(),get_current_ICOM()                    *
*   CALLING MODULES:     search_box()                            *
*                                                                *
*   AUTHOR:  Donghak Jung                                        *
*   HISTORY:                                                     *
*                                                                *
*****************************************************************/

void
store_diagram(fp,bbox)
FILE    *fp;
struct box_struct *bbox;
{
extern itoa(),get_current_ICOM();
extern struct header_struct *header_rootnode;
extern struct box_struct *box;
struct text_line_struct *tl;
char buf[DESCRIPTION_LINE_LENGTH+1];
char start2[20];

/* NAME */
```

```
fprintf(fp,"confirmed([%s,is,in_data]).\n",
                        bbox->name.text_string);

/* NUMBER */

itoa(bbox->number,buf);

fprintf(fp,"confirmed([%s,number_is,%s]).\n",
                          bbox->name.text_string,buf);

box = bbox;     /* "box" needed for get_current_ICOM()*/

/* INPUTS */

tl = (struct text_line_struct *)get_current_ICOM('I');

while(tl != NULL)
        {
        fprintf(fp,"confirmed([%s,input_is,%s]).\n",
                                  bbox->name.text_string,
                                            tl->text_line);
        tl = tl->next;
        }

/* OUTPUTS */

tl = (struct text_line_struct *)get_current_ICOM('O');

if (tl == NULL)
       {
        fprintf(fp,"confirmed([%s,output_is,' ']).\n",
                    bbox->name.text_string);
       }
   else
       {
        while(tl != NULL)
          {
           fprintf(fp,"confirmed([%s,output_is,%s]).\n",
                        bbox->name.text_string,
                       tl->text_line);
           tl = tl->next;
```

```
            }
        }

/* CONTROLS */

tl = (struct text_line_struct *)get_current_ICOM('C');

if (tl == NULL)
        {
         fprintf(fp,"confirmed([%s,control_is,' ']).\n]",
                        bbox->name.text_string);
        }
   else
        {
         while(tl != NULL)
           {
            fprintf(fp,"confirmed([%s,control_is,%s]).\n",
                                bbox->name.text_string,
                        tl->text_line);
            tl = tl->next;
           }
        }

/* MECHANISMS */

tl = (struct text_line_struct *)get_current_ICOM('M');

while(tl != NULL)
        {
        fprintf(fp,"confirmed([%s,mechanism_is,%s]).\n",
                        bbox->name.text_string,tl->text_line);
        tl = tl->next;
        }

return;
}
```

```
/*****************************************************************
*                                                                *
*   DATE:     2  July 1988                                       *
*   VERSIOIN: 1.0                                                *
*                                                                *
*   NAME:              search_box()                              *
*   MODULE NUMBER:                                               *
*   DESCRIPTION:                                                 *
*   This purpose of this module is to walk through               *
*   all the activity boxes.                                      *
*                                                                *
*   ALGORITHM:                                                   *
*   PASSED VARIABLES:         fp                                 *
*                                                                *
*   RETURNS:                                                     *
*   GLOBAL VARIABLES USED:    box_rootnode                       *
*   GLOBAL VARIABLES CHANGED:                                    *
*   FILES READ:                                                  *
*   FILES WRITTEN:                                               *
*   HARDWARE INPUT:                                              *
*   HARDWARE OUTPUT:                                             *
*   MODULES CALLED:           store_diagram()                    *
*   CALLING MODULES:          init_translation()                 *
*                                                                *
*   AUTHOR:  Donghak Jung                                        *
*   HISTORY:                                                     *
*                                                                *
*****************************************************************/

int
search_box(fp)
FILE    *fp;
{
extern struct box_struct *box_rootnode;
struct box_struct *bbox;

bbox= box_rootnode;
while(bbox != NULL)
        {
        if(bbox->dd != NULL)
                {
                 store_diagram(fp,bbox);
```

```
                }
        bbox = bbox->next;
        }
return(MYTRUE);
}
```

```
/*****************************************************************
*                                                                *
*   DATE:      2  July 1988                                      *
*   VERSION: 1.0                                                 *
*                                                                *
*   NAME:              handle_file()                             *
*   MODULE NUMBER:                                               *
*   DESCRIPTION:                                                 *
*   The purpose of this module is to maintain a pointer to a     *
*   file.  It may be set and later recalled                      *
*                                                                *
*   ALGORITHM:                                                   *
*   PASSED VARIABLES:          fp,command                        *
*                                                                *
*   RETURNS:    file_pointer                                     *
*   GLOBAL VALIABLES USED:                                       *
*   GLOBAL VALIABLES CHANGED:                                    *
*   FILES READ:                                                  *
*   FILES WRITTEN:                                               *
*   HARDWARE INPUT:                                              *
*   HARDWARE OUTPUT:                                             *
*   MODULE CALLED:                                               *
*   CALLING MODULES:    init_translation()                       *
*                                                                *
*   AUTHOR: Donghak Jung                                         *
*   HISTORY:                                                     *
*                                                                *
*****************************************************************/

FILE *
handle_file(fp,command)
FILE *fp;
char command[5];
{
static FILE *file_pointer;

if((strcmp(command,"set"))==0)
{
file_pointer = fp;
}
else if((strcmp(command,"get"))==0)
{
```

```
return(file_pointer);
}

return(NULL);
}
```

```
/*****************************************************************
*                                                                *
*   DATE:    2  July 1988                                        *
*   VERSION: 1.0                                                 *
*                                                                *
*   NAME:          init_translation()                            *
*   MODULE NUMBER:                                               *
*   DESCRIPTION:                                                 *
*   The purpose of this module is to get the file name from the  *
*   in which to save the predicates files.                       *
*                                                                *
*   ALGORITHM:                                                   *
*   PASSED VARIABLES:                                            *
*                                                                *
*   RETURNS:        PANEL_NONE (Sunview variable)                *
*   GLOBAL VARIABLES USED:                                       *
*   GLOBAL VARIABLES CHANGED:                                    *
*   FILES READ:                                                  *
*   FILES WRITTEN:                                               *
*   HARDWARE INPUT:                                              *
*   HARDWARE OUTPUT:                                             *
*   MODULES CALLED: put_message(),fix_input(),                   *
*         disable_input_window(),handle_file(),search_box()      *
*   CALLING MODULES: translate_diagram()                         *
*                                                                *
*   AUTHOR:  Donghak Jung                                        *
*   HISTORY:                                                     *
*                                                                *
*****************************************************************/

init_translation()
{

extern put_message(),disable_input_window();
extern fix_input();
char name[FILE_NAME_LENGTH+1],name2[FILE_NAME_LENGTH+5];
FILE *fp,*fopen();

/* get the user input */
strcpy(name,(char *)panel_get_value(input_item));
fix_input(name);
```

```
if(strcmp(name,"")==0)
{
put_message(1,"OPERATION ABORTED--
            NO FILE NAME RECEIVED--Make another selection");
disable_input_window();
return(PANEL_NONE);
}

strcpy(name2,name);
strcat(name2,".pro");

if((fp = fopen(name2,"w")) == NULL)
{
put_message(1,
                "Unable to open the predicate file-- ABORT");
disable_input_window();
}
else
{
disable_input_window();
handle_file(fp,"set");
        put_message(1,"Translating & Saving .....");
        fp = handle_file(fp,"get");
        search_box(fp);
        fclose(fp);
        put_message(1,
   "Translating & Saving are done... Make another selection ");

}
return(PANEL_NONE);
}
```

```
/*****************************************************************
*                                                                *
*   DATE:    2  July 1988                                        *
*   VERSION: 1.0                                                 *
*                                                                *
*   NAME:          translate_diagram()                           *
*   MODULE NUMBER:                                               *
*   DESCRIPTION:                                                 *
*   The purpose of this module is to prompt the user an set      *
*   up the Sunview environment to save the predicates in a       *
*   file.                                                        *
*                                                                *
*   ALGORITHM:                                                   *
*   PASSED VARIABLES:                                            *
*                                                                *
*   RETURNS:                                                     *
*   GLOBAL VARIABLES USED:                                       *
*   GLOBAL VARIABLES CHANGED:                                    *
*   FILES READ:                                                  *
*   FILES WRITTEN:                                               *
*   HARDWARE INPUT:                                              *
*   HARDWARE OUTPUT:                                             *
*   MODULES CALLED: put_message(),enable_input_window(),         *
*                   my_window_set(),init_translation()           *
*   CALLING MODULES: check_syntax()                              *
*                                                                *
*   AUTHOR:  Donghak Jung                                        *
*   HISTORY:                                                     *
*                                                                *
*****************************************************************/

 void
 translate_diagram(win,event,arg)
 Window window;
 Event *event;
 caddr_t arg;
 {
 extern put_message(),enable_input_window(),my_window_set();
 extern null_proc();

 if(event_is_up(event)) return;
 switch event_id(event)
```

```
{
case MS_LEFT:

my_window_set(null_proc);
enable_input_window();
panel_set(input_item,
PANEL_VALUE_STORED_LENGTH,FILE_NAME_LENGTH,
PANEL_NOTIFY_PROC,init_translation,
0);
put_message(1,"Enter the file name and hit RETURN");
break;

case MS_RIGHT:

my_window_set(null_proc);
put_message(1,"OPERATION ABORTED -- Make another selection");
break;
}
return;
}
```

```
/*****************************************************************
*                                                                *
*   DATE:    2  July 1988                                        *
*   VERSION: 1.0                                                 *
*                                                                *
*   NAME:          check_syntax()                                *
*   MODULE NUMBER:                                               *
*   DESCRIPTION:                                                 *
*   The purpose of this module is to initialize the user         *
*   selection for the syntax check of a given SA diagram.        *
*                                                                *
*   ALGORITHM:                                                   *
*   PASSED VARIABLES:                                            *
*                                                                *
*   GLOBAL VARIABLES USED:                                       *
*   GLOBAL VARIABLES CHANGED:                                    *
*   FILES READ:                                                  *
*   FILES WRITTEN:                                               *
*   HARDWARE INPUT:                                              *
*   HARDWARE OUTPUT:                                             *
*   MODULES CALLED:  translate_diagram()                         *
*   CALLING MODULES: make_windows()                              *
*                                                                *
*   AUTHOR:  Donghak Jung                                        *
*                                                                *
*****************************************************************/

 void
 check_syntax()
 {

 extern put_message(),my_window_set();
 extern my_move_cursor();

 put_message(1,"CHECK SYNTAX: |L| to check |R| to abort");
 my_move_cursor(INIT_LOC_X,INIT_LOC_Y);
 my_window_set(translate_diagram);

 return;

}
```

```
/*****************************************************************/
/*                                                               */
/*    DATE:    10  July  1988                                    */
/*    VERSION: 1.0                                               */
/*                                                               */
/*    NAME:    BC3                                               */
/*    DESCRIPTION:                                               */
/*    The perpose of this module is to provide an inference      */
/*    engine for the syntax checker.  BC3 provides a shell for   */
/*    backward chaining expert systems.                          */
/*                                                               */
/*    OPERATING SYSTEM:  MS-DOS                                  */
/*    LANGUAGE:          PROLOG-1                                */
/*    CONTENTS:            *                                    */
/*                                                               */
/*    AUTHOR: DR. Frank M. Brown                                 */
/*    HISTORY:                                                   */
/*****************************************************************/

/*****************************************************************/
/*                                                               */
/*                            BC4                                */
/*                                                               */
/*       A shell for backward-chaining expert systems.           */
/*                                                               */
/*  Each item of knowledge is represented by a triple (i.e.,     */
/*  a three-element list of the form [Object,Attribute,Value].   */
/*  An associated rule-base supplies the following data:         */
/*                                                               */
/*  1. A goals-statement, in the form of a list of triples to    */
/*     be solved in sequence. The solved triples are printed     */
/*     by the shell.                                             */
/*  2. A collection of if-then rules for triples.                */
/*  3. A collection of facts, i.e., triples asserted as known    */
/*     a priori.                                                 */
/*  4. A collection of 'askable' triples, indicating the forms   */
/*     of triples whose values may be obtained from the user.    */
/*  5. A collection of 'keep' triples, indicating the form of    */
/*     the triples not to be erased from working memory at       */
/*     the beginning of a new session.                           */
```

```
/*                                                                  */
/*  Each item of knowledge stored in working memory is of the       */
/*  form confirmed([Obj,Attr,Val]) or denied([Obj,Attr,Val]).       */
/*                                                                  */
/*  To use the system, load BC4, load the appropriate rule-         */
/*  base and type 'start.' Because BC4's operator-defini-           */
/*  tions are used by the rule-bases, BC4 must load first.          */
/*                                                                  */
/******************************************************************/



/*------------------- OPERATOR DEFINITIONS --------------------*/
/*                                                              */
/*  The operators defined below enable the rules in the know-   */
/*  ledge-base to be expressed in a form more readable than     */
/*  the standard (prefix) form.                                 */
/*                                                              */
/*--------------------------------------------------------------*/

?- op(250, xfx, :).
?- op(245, xfx, then).
?- op(240, fx,  if).
?- op(235, xfx, derived_from).
?- op(230, xfy, or).
?- op(225, xfy, and).

/*------------------------- START ------------------------------*/
/*                                                              */
/*  The procedure 'start' begins by erasing from working mem-   */
/*  ory all 'confirmed' and 'denied' clauses, except those      */
/*  clauses protected by 'keep' from erasure. The list of       */
/*  goal-triples is then read from the rule-base and solved in  */
/*  turn by 'solve'. A trace is maintained of the back-         */
/*  chaining search-tree generated in solving the goals. When   */
/*  the last of the goal-triples is solved, the values of all   */
/*  goals, except those solved by asking the user directly,     */
/*  are displayed; the trace is also displayed, if requested,   */
/*  as a "how" explanation of the solution.                     */
/*                                                              */
/*--------------------------------------------------------------*/

start :-
```

```
  (ask_about_loading_wm, ! ; scrub_wm),main_start.

scrub_wm :-
  ( confirmed(Triple),                  /* Erase all working-mem- */
    not(keep:Triple),                   /* ory elements not pro-  */
    retract(confirmed(Triple))          /* tected by 'keep' state-*/
  ;                                     /* ments in the knowledge-*/
    denied(Triple),                     /* base.                  */
    not(keep:Triple),
    retract(denied(Triple)) ),
  fail.
scrub_wm.

main_start :-
  retract_all(why_trace(_)),            /* Erase the "why" trace. */
  goals: Goals,                         /* Find the goal-triples, */
  prefix(Goals,Prefixed_goals),         /* prefix each of them    */
  reverse(Prefixed_goals,Goal_list),    /* with the word 'goal',  */
                                        /* & reverse their order. */
  solve(Goals,[],Part_trace),           /* Satisfy all of the     */
  !,nl,                                 /* goals and then put the */
  append(Goal_list,Part_trace,Trace),
                                        /* list of goals at the   */
                                        /* front of the "how"     */
  ask_about_trace(Trace),               /* trace. Supply a "how"  */
  ask_about_saving_wm.                  /* explanation on request.*/

main_start :-                           /* If all triples can't   */
  nl,                                   /* be solved, announce it.*/
  write('I can''t solve this problem.'),nl.


/*-------------------------- SOLVE --------------------------*/
/*                                                           */
/*  The predicate 'solve(Goals,Trace,New_trace)' means that  */
/*  Goals is a list of goals (expressed as triples), and that */
/*  Trace and New_trace are, respectively, the trace-lists be- */
/*  and after solution of the goal at the head of the goal-  */
/*  list. The procedure 'solve' solves each of the goals in  */
/*  turn. The first step in solving a goal is to erase the   */
/*  "why" trace and to initialize it with that goal. Thus each */
/*  goal is solved with a separate "why" trace. As each rule */
```

```
/*  is encountered in descending through the search-tree for a */
/*  given goal, that rule is added to the front of the "why"   */
/*  trace.                                                     */
/*                                                             */
/*-------------------------------------------------------------*/

solve([],Trace,Trace).
solve([Goal|Others],Trace,New_trace) :-
  retract_all(why_trace(_)),          /* Initialize the "why"   */
  asserta(why_trace([goal:Goal])),    /* trace.                 */
  is_known(Goal,Trace,Trace1),
  ( confirmed(Goal),!                 /* Write each triple as   */
  ;                                   /* it's solved, but don't */
    nl,write_triple(Goal),nl ),       /* write a triple that's  */
  solve(Others,Trace1,New_trace).     /* been told explicitly   */
                                      /* by the user.           */

write_triple([Obj,Attr,Val]) :-
  writelist([Obj,' ',Attr,' ',Val,'.']).

ask_about_trace(Trace) :-
  write('Do you wish to see how this answer was arrived at? '),
  read(Reply),
  ( means(Reply,yes), !,
    write_trace(Trace)
  ;
    true ).

ask_about_loading_wm :-
  write('Do you wish to load from a working-memory file?  '),
  read(Reply),nl,
  Reply = y,
  load_working_memory.

ask_about_saving_wm :-
  write('Do you wish to save working memory in a file?  '),
  read(Reply),nl,
  ( Reply = y,
    save_working_memory, !
  ;
    true).
```

```
prefix([],[]).
prefix([Goal|Goals],[goal:Goal|Prefixed_Goals]) :-
  prefix(Goals,Prefixed_Goals).


/*------------------------- IS_KNOWN -------------------------*/
/*                                                            */
/*  The 'is_known' procedure maintains a trace of the path of */
/*  the solution-tree leading to the triple currently under   */
/*  consideration. 'is_known(Triple,Trace,New_trace)' means   */
/*  that if reasoning to a certain point has been recorded in */
/*  the list 'Trace', then the additional triple 'Triple' is  */
/*  known via reasoning recorded by the list 'New_trace'.     */
/*                                                            */
/*------------------------------------------------------------*/

/* A triple is not known if it has been denied by the user.   */

is_known(Triple,Trace,Trace) :-
  denied(Triple),
  !,
  fail.

/* The triple [O,A,V] is known if it is a fact in the rule    */
/* base. If the current trace includes an entry that [O,A,V]  */
/* is a fact, then leave the trace alone; otherwise, augment  */
/* the trace with such an entry.                              */

is_known([O,A,V],Trace,Trace) :-
  member(fact:[O,A,V],Trace),
  !.

is_known([O,A,V],Trace,[fact:[O,A,V]|Trace]) :-
  fact: [O,A,V],
  !.

/* A triple is known if it has been confirmed by the user.    */

is_known(Triple,Trace,Trace) :-
  member(was_told:Triple,Trace),
  !.
```

```
is_known(Triple,Trace,[was_told:Triple|Trace]) :-
  confirmed(Triple),
  !.

/* A triple [X,P,Y] is known if the Prolog goal P(X,Y) suc-    */
/* ceeds, either because P is a built-in predicate, or because */
/* the rule-base has prolog-code defining P. The triple        */
/* [2,member,[1,2]], for example, is converted into the goal   */
/* member(X,[1,2]), which is then executed by Prolog. To keep  */
/* non-Prolog-programmers out of trouble, the triple [X,is,Y]  */
/* is trapped so that it will not be executed as an arithmetic */
/* statement. The triple [X,:=,Y] is interpreted as Prolog's   */
/* arithmetic or assignment goal, X is Y.                      */

is_known([Obj,Attr,Val],Trace,Trace) :-
  member(solved:[Obj,Attr,Val],Trace),
  !.

is_known([Obj,Attr,Val],Trace,[solved:[Obj,Attr,Val]|Trace]) :-
  nonvar(Attr),           /* To avoid an error-halt.           */
  (
    Attr == :=, !,
    Obj is Val            /* Interpret ':=' as Prolog's 'is'.  */
  ;
    Attr == =, !,         /* Interpret '=' as an exact match.  */
    Obj  == Val
  ;
    not (Attr == is),     /* Interpret everything else, except */
    T =.. [Attr,Obj,Val], /* 'is', as a functor on a two-place */
    T, !                  /* predicate to be solved as a goal. */
  ).



/* A triple is known if it is the head of a rule and the con-  */
/* ditions of the rule are satisfied. We put a rule that we    */
/* encounter at the head of the "why" trace, erasing any du-   */
/* plicates of the rule that are already in the "why" trace.   */
/* The "why" trace is maintained in the database, in a clause  */
/* of the form 'why_trace(<List of goals and rules>)'. This    */
/* differs from the "how" trace, which is handed as an argu-   */
/* ment from goal to goal.                                     */
```

```
is_known(Triple,Trace,[was_proved:[Triple,Rule]|Trace]) :-
  member(Rule: Triple derived_from Conds,Trace),
  !.

is_known(Trpl,Trc,[Rule: Trpl derived_from Conds|Trc1]) :-
  Rule: if Conds then Trpl,
  why_trace(Why_trace),
  remove(Rule: Trpl derived_from Conds,Why_trace,Part_why),
  append([Rule: Trpl derived_from Conds],Part_why,New_why),
  retract(why_trace(_)),
  asserta(why_trace(New_why)),
  is_known(Conds,Trc,Trc1),
  !.

/* A condition involving "and", "or", or "not" is known if its */
/* parts are known in suitable combinations.                   */

is_known(Triples1 and Triples2,Trace,Trace2) :-
  is_known(Triples1,Trace,Trace1),
  is_known(Triples2,Trace1,Trace2).

is_known(Triples1 or Triples2,Trace,Trace1) :-
  is_known(Triples1,Trace,Trace1).
is_known(Triples1 or Triples2,Trace,Trace2) :-
  is_known(Triples2,Trace,Trace2).

is_known(not Triple,Trace,[confirmed_not:Triple|Trace]) :-
  not is_known(Triple,Trace,Trace1).

/* A triple is known if (a) the rule-base classifies it as     */
/* "askable" and if (b) the user confirms it. The user may     */
/* request a "why" explanation before responding to the ques-  */
/* tion.                                                       */

is_known([O,A,V],Trace,Trace) :-
  member(was_told:[O,A,V],Trace),
  !.

is_known([O,A,V],Trace,[was_told:[O,A,V]|Trace]):-
  askable: [O,A,_],
  ask_about([O,A,V]),
  !,
```

```
  confirmed([O,A,V]).


/*------------------------ ASK_ABOUT ------------------------*/

ask_about([Obj,Attr,Val]) :-
  var(Val),
  !, nl,
  writelist([Obj,' ',Attr,'? ']),nl,
  askable: [Obj,Attr,Legal_values],
  write('Legal values: '), write(Legal_values), nl,
  write('> '), read(Reply),
  ( (
      means(Reply,why),              /* If the user responds    */
      explain_why([Obj,Attr,Val]),   /* with 'why.', give him   */
      !,                             /* an explanation.         */
      ask_about([Obj,Attr,Val])
    )
  ;

    (
      atomic(Legal_values)
    ;
      member(Reply,Legal_values)
    ),
    !,
    assertz(confirmed([Obj,Attr,Reply]))
  ;
    write('Please re-enter your reply.'),nl,
    ask_about([Obj,Attr,Val])
  ).

ask_about([Obj,Attr,Val]) :-
  nl,
  writelist([Obj,' ',Attr,' ',Val,'? (yes./no./why.)']),
  nl,write('> '),read(Reply),
  (
    means(Reply,yes),
    assertz(confirmed([Obj,Attr,Val])), !
  ;
    means(Reply,no),
    assertz(denied([Obj,Attr,Val])), !
```

```
  ;
    means(Reply,why),
    explain_why([Obj,Attr,Val]),
    !,
    ask_about([Obj,Attr,Val])
  ;
    write('Please re-enter your reply.'),nl,
    ask_about([Obj,Attr,Val])
  ).

means(Reply,yes) :-
  member(Reply,[y,yes]).
means(Reply,no) :-
  member(Reply,[n,no]).
means(Reply,why) :-
  member(Reply,[why,w]).


/*------------------------ EXPLAIN_WHY ------------------------*/

explain_why(Triple) :-
  why_trace(Why_trace),
  write('Because:'),nl,
  justify(Triple,Why_trace).

justify(Triple,Why_trace) :-
  member(goal:Goal,Why_trace),
  Triple = Goal,
  writelist(['This will satisfy the goal ',Goal]),nl,
  nl,
  !.
justify(Triple,Why_trace) :-
  member(R:Head derived_from Cs,Why_trace),
  among(Triple,Cs),
  remove(R:Head derived_from Cs,Why_trace,New_trace),
  writelist(['I can use ',Triple]),nl,
  list_known_triples(Cs),
  writelist(['     to help satisfy ',R,': ',Head]),nl,nl,
  justify(Head,New_trace).

list_known_triples(Cs) :-
  among(Triple,Cs),
  (
```

```
    confirmed(Triple)
  ;
    fact: Triple
  ),
  writelist(['     knowing ',Triple]),nl,
  fail.
list_known_triples(_).

among(Triple,Conditions) :-
  Triple = Conditions.
among(Triple, First_triple and Other_conditions) :-
  Triple = First_triple,!
  ;
  among(Triple,Other_conditions).
among(Triple, First_triple or Other_conditions) :-
  Triple = First_triple,!
  ;
  among(Triple,Other_conditions).


/*------------------------ WRITE_TRACE ------------------------*/

write_trace([]) :-
  nl.
write_trace([goal:Triple|Rest]) :-
  !,
  write('GOAL:   '),write(Triple),nl,
  write_trace(Rest).
write_trace([fact:Triple|Rest]) :-
  !,
  write('FACT:   '),write(Triple),nl,
  write_trace(Rest).
write_trace([solved:Triple|Rest]) :-
  !,
  write('SOLVED: '),write(Triple),nl,
  write_trace(Rest).
write_trace([was_told:Triple|Rest]) :-
  !,
  write('TOLD:   '),write(Triple),nl,
  write_trace(Rest).
write_trace([confirmed_not:Triple|Rest]) :-
  !,
```

```
  write('CONTRADICTED: '),write(Triple),nl,
  write_trace(Rest).
write_trace([was_proved:[Triple,Rule]|Rest]) :-
  !,
  write('PROVED: '),write(Triple),write(' using '),write(Rule),nl,
  write_trace(Rest).
write_trace([Rule: Triple derived_from Conditions|Rest]) :-
  !,
  writelist([Rule,': ',Triple,'  Was Derived From']),nl,
  write_conditions(Conditions),
  write_trace(Rest).
write_trace([X|Rest]) :-
  write(X),nl,
  write_trace(Rest).

write_conditions([X,Y,Z]) :-
  tab(8),write([X,Y,Z]),nl.
write_conditions(not [X,Y,Z]) :-
  tab(4),write('NOT '),write([X,Y,Z]),nl.
write_conditions([X,Y,Z] and Conditions) :-
  tab(8),write([X,Y,Z]),write(' AND'),nl,
  write_conditions(Conditions).
write_conditions(not [X,Y,Z] and Conditions) :-
  tab(4),write('NOT '),write([X,Y,Z]),write(' AND'),nl,
  write_conditions(Conditions).
write_conditions([X,Y,Z] or Conditions) :-
  !,
  tab(8),write([X,Y,Z]),write(' OR'),nl,
  write_conditions(Conditions).
write_conditions(Conditions1 or Conditions2) :-
  write_conditions(Conditions1),tab(8),write('OR'),nl,
  write_conditions(Conditions2).
write_conditions(not [X,Y,Z] or Conditions) :-
  tab(4),write('NOT '),write([X,Y,Z]),write(' OR'),nl,
  write_conditions(Conditions).



/*------------------------ FILE I/O -------------------------*/

save_working_memory :-
  write('Please supply a filename: '),
  read(Filename),nl,
```

```
  tell(Filename),
  save_wme,
  told.

save_wme :-
  confirmed(Triple),
  writeq(confirmed(Triple)),write('.'),nl,
  fail.
save_wme :-
  denied(Triple),
  writeq(denied(Triple)),write('.'),nl,
  fail.
save_wme.

load_working_memory :-
  write('Please supply a filename: '),
  read(Filename),nl,
  retract_all(confirmed(_)),
  retract_all(denied(_)),
  see(Filename),
  loadfile,
  write('Contents of working memory:'),nl,nl,
  list_working_memory,
  seen.

loadfile :-
  read(Term),
  load(Term).

load(end_of_file) :-
  !.
load(Term) :-
  not Term =.. [confirmed,_],
  not Term =.. [denied,_],
  !,
  write('Not a legal file of working-memory elements...'),nl,nl,
  retract_all(confirmed(_)),
  retract_all(denied(_)).
load(Term) :-
  assertz(Term),
  loadfile.
```

```
list_working_memory :-
  confirmed(Triple),
  write(confirmed(Triple)),write('.'),nl,
  fail.
list_working_memory :-
  denied(Triple),
  write(denied(Triple)),write('.'),nl,
  fail.
list_working_memory.


/*-------------------- UTILITY PROCEDURES ----------------------*/

writelist([]).
writelist([X|L]) :-
  write(X),
  writelist(L).

member(X,[X|_]).
member(X,[_|L]) :-
  member(X,L).

append([],L,L).
append([X|L],M,[X|N]) :-
  append(L,M,N).

reverse([],[]).
reverse([X|L],M) :-
  reverse(L,N),
  append(N,[X],M).

remove(_,[],[]).
remove(X,[X|L],M) :-
  !,
  remove(X,L,M).
remove(X,[Y|L],[Y|M]) :-
  remove(X,L,M).

retract_all(X) :-
  not not retract(X),
  retract_all(X).
retract_all(X) :-
```

```
  not not retract((X :- Y)),
  retract_all(X).
retract_all(_).

wm :-
  listing(confirmed),
  listing(denied).

reset :-
  retract_all(confirmed(_)),
  retract_all(denied(_)).

again :-
  write('Consulting bc3.pro'),nl,
  reconsult('bc3.pro').

why :-
  why_trace(Trace),
  write_trace(Trace).

?- write('Type  ''start.''   to begin.'),nl,nl,
   write('Answer all questions using lower case,'),nl,
   write('ending with a period.'),nl.
```

Knowledge base

```
/*************************************************************/
/*                                                           */
/*  DATE: 5 July 1988                                        */
/*  VERSION: 1.0                                             */
/*                                                           */
/*  DESCRIPTION:                                             */
/*  This file contains syntax rules for the syntax           */
/*  checker.                                                 */
/*                                                           */
/*  OPERATING SYSTEM: MS-DOS                                 */
/*  LANGUAGE: PROLOG-1                                       */
/*  CONTENTS:                                                */
/*                                                           */
/*  AUTHOR: Dong Hak jung                                    */
/*  HISTORY:                                                 */
/*                                                           */
/*************************************************************/

/*--------  goal -----*/

/* These lists of goal present the resulting message. */

  goals: [[goal_1, ' ', Message1],
          [goal_2, ' ', Message2],
          [goal_3, ' ', Message3]].


/*-------- askable ---*/


/* This "askable" statement needs to enter a box name.  */

askable: [boxname, is, 'type in the box name'].


/*-------- facts -------*/

/* The facts needs to enter a working memory file.       */


/*-------- rules ------*/
```

```
/*  Rules 1 through 4 determine whether a requested     */
/*  box name is contained in the facts data base or not.*/

rule1:   if [boxname, is, Box]
        and not [Box,is,in_data]

       then [program, should_be, stopped].

rule2:   if [program, should_be, stopped]

       then [goal_1,  ' ',  ': Requested box is not in the
                            data base.'].

rule3:   if [program, should_be, stopped]

       then [goal_2, ' ', ': Start again, please !!!'].

rule4:   if [program, should_be, stopped]

       then [goal_3, ' ', ' '].


/* If there is a box and the number of the box is empty,   */
     then  there is no number in the box.                  */

rule5:   if [boxname, is, Box]
        and [Box, number_is, ' ']

       then [goal_1, ' ', 'Error: There is no box number.'].


/* If there is a box and the name of the control is empty, */
     then  there is no control/name in the box.            */

rule6:   if [boxname, is, Box]
        and [Box, control_is, ' ']

      then [goal_2, ' ', 'Error: There is no control/name.'].

rule7:   if [boxname, is, Box]
```

```
        and not [Box, control_is, ' ']

       then [goal_2, ' ', 'Control is OK.'].


/* If there is a box and the output of the box is empty,   */
/*    then there is no output/name.                        */

rule8:   if [boxname, is ,Box]
        and [Box, output_is, ' ']

       then [goal_3, ' ', 'Error: There is no output/name.'].

rule9:   if [boxname, is, Box]
        and not [Box, output_is, ' ']

       then [goal_3, ' ', 'Output is OK.'].


/* If there is a box                                       */
/*    and the number of the box is less than 1             */
/*     or the number of the box is greater 6,              */
/*    then the box number is beyond the limit.             */

rule10:  if [boxname, is, Box]
        and [Box, number_is, Number]
        and not [Number, ==, ' ']
        and [YesNo, within_limit, Number]

       then [box_number_is, legitimate, YesNo].

            within_limit(YesNo,Number) :-
                      wlimit(Number,YesNo).
                      wlimit(N,yes) :- N > 0, N < 7, !.
                      wlimit(N,no)  :- N =< 0; N > 6.

rule11:  if [box_number_is, legitimate, YesNo]
        and [YesNo, ==, yes]

       then [goal_1, ' ', 'Box number is OK.'].

rule12:  if [box_number_is, legitimate, YesNo]
```

```
        and [YesNo, ==, no]

        then [goal_1, ' ', 'Error: Box number is beyond the
                limit.'].


/*---------------  end  ----------------------------*/
```

Appendix D. *User's Guide*

Descriptions

The SADT Validator is operated through the use of the SADT Edi
is assumed that user of the tool is familiar with the SADT method and
use. In addition, the user is forced to have knowledge of the UNIX enviro

System Requirements

Workstations: Sun and Z-248.

Software: SunView, Sunwindow environment, C, and Prolog-1 versic

Operation on Sun Workstation

1. Login to the Sun workstation by normal UNIX login procedure.
2. Enter "suntools".
3. Enter "SAtool".
4. Begin by selecting a menu option. Menus are displayed on the screen b
   the cursor inside one of the "ovals" above the diagram and clickin
   mouse button. The oval choices are:
   - RECALL DGM
   - EDIT DGM
   - EDIT DD
   - EDIT FPT
   - EDIT MISC
   - SAVE DGM
   - CHECK SYNTAX

A detailed guide for editing an SADT diagram is available in the user's manual of Johnson's thesis (8).

5. After drawing an SADT diagram, use the mouse to select "CHECK SYNTAX" oval to begin checking the SADT syntax of the diagram.
6. Enter the filename in the Input window. Then, <filename>.pro file is created in the current directory.
7. Exit the tool by selecting "QUIT" under the "EDIT MISC" oval.
8. From the suntool menu, select "Exit Suntools" by clicking the LEFT mouse button.
9. Enter "logout".

Operation on Z-248 Workstation

1. Go to a Z-248 workstation connected under AFITNET.
2. Enter "ftp <Sun workstation name>".
3. Login to the Sun workstation by normal UNIX login procedures.
4. Enter "get <fileneme>.pro".
5. Enter "bye".
6. Enter "prolog".
7. Enter "consult('bc4').".
8. Enter "consult('sarule').".
9. Enter "start.". Then, the message "Do you wish to load from working memory file ? " is displayed.
10. Enter "y.".
11. Enter "<filename>.pro.".

12. Enter the box name which you want to check. Then, the resulting messages of the syntax checking procedure are displayed. In addition, the message "Do you wish to see how this answer was arrived at ?" is displayed.

13. Enter "y." or "n.". If you enter "y.", then the message regarding the answer derived is displayed. In both cases, the message "Do you wish to save working memory in a file ?" is displayed.

14. Enter "y." or "n.". If you want to check other boxes of the SADT diagram, repeat steps 9 through 15.

15. In order to exit, enter "Ctrl C".

Example of Predicate File

This section presents an example of the predicate file translated from an SADT diagram. Figure D.1 shows an example of the predicate file translated from the SADT diagram shown in Figure D.2.

```
confirmed([box1,is,in_data]).
confirmed([box1,number_is,1]).
confirmed([box1,input_is,in1]).
confirmed([box1,input_is,in2]).
confirmed([box1,output_is,out2]).
confirmed([box1,control_is,con1]).
confirmed([box2,is,in_data]).
confirmed([box2,number_is,2]).
confirmed([box2,input_is,out2]).
confirmed([box2,input_is,in3]).
confirmed([box2,output_is,in2]).
confirmed([box2,output_is,out3]).
confirmed([box2,output_is,out4]).
confirmed([box2,control_is,con2]).
confirmed([box2,control_is,con3]).
confirmed([box3,is,in_data]).
confirmed([box3,number_is,3]).
confirmed([box3,input_is,in4]).
confirmed([box3,output_is,out5]).
confirmed([box3,output_is,con3]).
confirmed([box3,control_is,out4]).
```

Figure D.1. Example of Predicate File

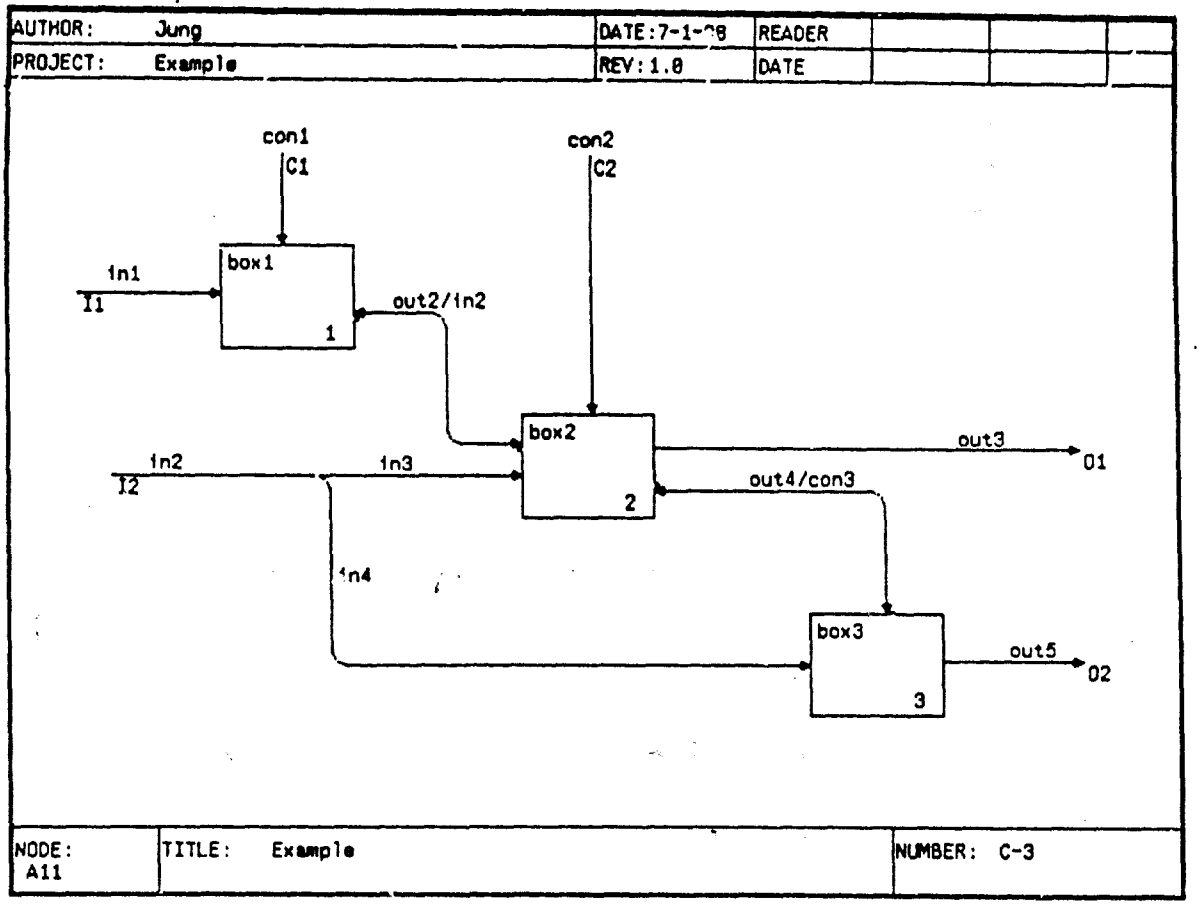


Figure D.2. Example of SADT Diagram

Appendix E. *Programmer's Guide*

The purpose of this appendix is to specify the procedure for generating the execution files for the programmers. The tool was implemented in two part, the translator and the syntax checker. Since the translator was implemented as an integral part of the SADT Editor on the Sun workstation, the file of the translator is contained in the files of the SADT Editor. The code file of the translator was named in "validator.c". The executable file was generated by using the UNIX "make" facility. Using this method, changes to the source files are tracked and recompiled as necessary before linking the files together. Figure E.1 is a copy of the file "Makefile." To use this file, the command "make" is typed at the system prompt, causing any needed compilations and then linking of this file.

The syntax checker was implemented on the Z-248 workstation. As mentioned in Chapter IV, the syntax checker was implemented in two sub-components, the inference engine and the knowledge base. The code file of the inference engine was named in "bc4.pro" and the code file of the knowledge base was named in "sarule.pro". The file "sarule.pro" needs a working memory file, which presents the facts, from user input. The name of the working memory file is created in "<filename>.pro" by the user input during the execution of the translator. The execution of the syntax checker is performed in the following steps:

1. A:> prolog
2. ?- consult('bc4').
3. ?- consult('sarule').
4. ?- start.

Under the environment of Prolog-1, all input should be ended with a period. Detailed operation is presented in Appendix D.

```
OBJECTS = main.o datadict.o messages.o boxfunctions.o
headerfunctions.o editboxfunc.o miscfunctions.o
addline.o figures.o endfuncs.o find.o morelinefuncs.o
linelabel.o moreddfuncs.o ddsearchfuncs.o savefuncs.o
fptfuncs.o sqglefuncs.o fnotefuncs.o moresave.o
screendump.o readfuncs.o dectalk.o session.o validator.o

HEADERS = globals.h

ALL = sad

CFLAGS = -O

LIBS = -lsuntool_p -lsunwindow_p -lpixrect -lm

sad : $(OBJECTS)
cc $(CFLAGS) $(OBJECTS) $(LIBS) -o SAtool
```

Figure E.1. Makefile Format

All of code files were stored in the directory " djung/validator" on the Sun workstation.

Bibliography

1. Agresti, William W. "The Conventional Software Life- cycle Model: its Evolution and Assumptions". *IEEE Tutorial: New Paradigms for Software Development*, 1986.
2. Agresti, William W. "What are the New Paradigms ?". *IEEE Tutorial: New Paradigms for Software Development*, 1986.
3. Boehm, B. W. "Software Engineering," *IEEE Transactions on Computer, Vol C-25*, 12:1226-1241 (December, 1976).
4. Bratko, Ivan. *Prolog Programming for Artificial Intelligence.* England: Addison-Wesley Company, 1986.
5. Greenspan, Sol J. *Requirement Modeling: A Knowledge Representation Approach to Software Requirement Definition.* PhD Thesis, Dept. of Computer Science, University of Toronto, (1984).
6. Greenspan, Sol J., Borgida, Alexander, and Mylopoulos, John. "A Requirement Modeling Language and its Logic," *Information Systems, Vol 11-1*, 9-23, (1986).
7. Hartrum, Thomas C. *Software Development Documentation Guidelines and Standards (Draft No.3a).* Air Force Institute of Technology Department of Engineering, Wright-Patterson AFB, OH, (September 26, 1986).
8. Johnson, Steven E. *A Graphics Editor for Structured Analysis with a Data Dictionary.* MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, (December 1987).
9. Lederman, A. *An Abstracted Bibliography on Programming Environments.* Dep. Elec. Eng. Comput. Sci., M. I. T., (June 1980).
10. Pressman, Roger S. *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill Book Company, 1986.
11. Rich, Elaine. *Artificial Intelligence.* McGraw-Hill Book Company, First edition, 1983.
12. Lubars, Mitchell D. and Mehdi T. Harandi. "Knowledge- Based Software Design Using Design Schemas," *Proceedings of the 9'th International Conference on Software Engineering*, Monterey, Cal., 253-262 (March 1987).
13. Ross, Douglas T. "Applications and Extensions of SADT," *IEEE Transactions on Computer*, 25-30. (April 1985).
14. Ross, Douglas T. "Structured Analysis (SA): A language for Communicating Ideas," *IEEE Transactions on Software Engineering, SE-3, No.1*:16-34 (January 1977).

15. Ross, Douglas T. and K. E. Schoman. "Structured Analysis for Requirement Definition," *IEEE Transactions on Software Engineering, SE-3, No.1*:6-15 (Januarv. 1977).

16. Softech Inc. *An Introduction to SADT Structured Analysis and Design Technique.* Softech Report 9022- 78R, Waitham, MA, 1976.

17. Teichroew, D., and E. A. Hershey. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions on Software Engineering, SE-3, NO.1*:41-48, January, 1977.

18. Teitelbaum, Tim. *The Cornell Programming Synthesizer: A Tutorial Introduction.* Cornell Univ. Technical Report TR 79-381, (January 1980).

19. Teitelbaum, Tim and Thomas Reps, *The Cornell Programming Synthesizer: A Syntax-Directed Programming Environment.* Cornell Univ. Technical Report TR 80-421, (May 1980).

20. UM 170133010. *Interim AUTOIDEF System User's Reference Manual.* Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB OH, 1982.

21. Urscheler James W. *Design of a Requirement Analysis Design Tool Integrated with a Data Dictionary in a Distributed Software Development Environment.* MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, (June 1986).

22. Xenologic Inc. *Xenologic Prolog User Manual.* Newark, CA, 1988.

Vita

Major Dong Hak Jung was born on [redacted] He graduated from SungDong Mechanical High School in 1976 and attended R.O.K. Air Force Academy where he received a Bachelor of Engineering degree with a major in Electrical Engineering in April 1980. He entered active duty in April 1980 and was assigned to R.O.K. Air Force Headquarters in Seoul where served as Programmar, Central Computer center, until Febuary 1982. He then attended Seoul National University and received a Bachelor of Science Degree with emphasis in Computer Science and Statistics in Febuary 1984. Upon graduation, he again served at the Headquarter of the Air Force in Seoul as System Programmer, Centeral Computer center. He entered the school of Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, in May 1986.

[redacted]

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: Unclassfied	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public relese Distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/88S-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: OSD/SDIO; 8b. OFFICE SYMBOL (If applicable): S/BM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): Pentagon, Washington, DC 20301-7100	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): DESIGN OF A SYNTAX VALIDATION TOOL FOR REQUIREMENTS ANALYSIS USING STRUCTURED ANALYSIS AND DESIGN TECHNIQUE (SADT)

12. PERSONAL AUTHOR(S): Dong Hak Jung, Major Republic of Korea Air Force

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1988, July	105

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software Engineering, Formalization, Knowledge-Based System, Graphics Tool
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis investigation presents the prototype development of a validation tool for checking the syntax of Structured Analysis and Design Technique (SADT) method.

The tool provides the requirements analyst and the designer with an environment for checking the SADT syntax of a diagram.

The tool was implemented using a knowledge-based system technique The syntax checking process permits the extension of the tool to the syntax/semantics knowledge representation of SADT methodology.

Approved for release in Accordance with AFR 190-1
W. Premerlani 12 Jan 1989

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	Unclassfied

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Dr. Gary B. Lamont	513-256-4279	AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE